Simple, Fast, Scalable, and Reliable Multiprocessor Algorithms

by

Siddhartha Visveswara Jayanti
సిద్ధార్థ విశ్వేశ్వర జయంతి
सिद्धार्थ विश्वेश्वर जयन्ति

B.S.E., Princeton University (2017)
S.M., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

Author .....
Department of Electrical Engineering and Computer Science
November 27, 2022

Certified by .....
Julian Shun
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Simple, Fast, Scalable, and Reliable Multiprocessor Algorithms

by

Siddhartha Visveswara Jayanti

సిద్ధార్థ విశ్వేశ్వర జయంతి

सिद्धार्थ विश्वेश्वर जयन्ति

Submitted to the Department of Electrical Engineering and Computer Science
on November 27, 2022, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis, I identify simplicity, speed, scalability, and reliability as four core design goals for multiprocessor algorithms, and design and analyze algorithms that meet these goals.

I design the first scalable algorithm for concurrent union-find. Our algorithm provides almost-linear speed-up, performing just $\Theta\left(m \cdot \left(\log\left(\frac{np}{m}+1\right) + \alpha\left(n, \frac{m}{np}\right)\right)\right)$ work when p processes execute a total of m operations on an instance with n nodes. I furnish the algorithm with a rigorous, machine-verified proof of correctness, and prove that its work-complexity is optimal amongst a class of symmetric algorithms, which captures the complexities of all known concurrent union-find algorithms. The algorithm is lightning quick in practice: it has improved the state-of-the-art in model checking [23] and spatial clustering [208], and is the *fastest* algorithm for computing connected components on both CPUs and GPUs [51, 95].

I introduce *concurrent fast arrays*, which are linearizable wait-free arrays that support all operations, *including initialization*, in just constant time. As an application, I design the first fixed-length *fast hash table*, which supports constant time initialization, insertions, and queries.

I define సామాన్య జాగృతి (*generalized wake-up*), which generalizes the information propagation problem called wake-up. I prove fundamental hardness results about this problem, and through reductions, show that any linearizable queue, stack, priority queue, counter, or union-find object's work complexity must increase with process count; these lower bounds are robust to both randomization and amortization. This thesis includes the original results in Telugu with Sanskrit abstract, along with their English translation.

I design optimal complexity locks for real-time and persistent memory systems. Our *abortable queue lock* is the first abortable lock to achieve $O(1)$ amortized RMR complexity for both cache-coherent (CC) and distributed shared memory (DSM) systems. It additionally provides "abortable first-come-first-served" fairness and supports "fast aborts". Our *recoverable queue lock* is the first recoverable lock to achieve the optimal $O(\log p / \log\log p)$ worst-case RMR complexity on both CC and DSM persistent memory systems. Both locks are innovations on our newly devised standard lock, whose design simplifies and unifies several previously known techniques.

This thesis also emphasizes rigorous guarantees for concurrent algorithms. I devise a novel universal, sound, and complete "tracking" technique for proving linearizable and strong linearizable correctness of concurrent algorithms. My collaborators and I have used this technique to give machine-verified proofs of correctness for multicore queue, union-find, and snapshot algorithms.

Finally, I prove and experimentally validate that asynchronous "HOGWILD!" Gibbs Sampling, a technique born from machine learning practice, can be used to accurately estimate expectations of polynomial and other statistics of graphical models satisfying Dobrushin's condition.

Thesis Supervisor: Julian Shun
Title: Professor of Electrical Engineering and Computer Science

Acknowledgment

రామేశ్వరస్తుతి

అసురాపహారసత్కారక్షక సత్కావర్తనజీవనహర్షిత
క్షీరాబ్ధిజాస్వీకరశ్రీధర శ్రీకరవరదాసదాసోహం ||

Foremost, I thank my Ph.D. advisor Julian Shun. When I met Julian in my first few days at MIT, I was embarking on a new line of research into statistical learning, and I had little idea then, that I would end up finishing my Master's thesis at the intersection of algorithms and economics and soon thereafter pursue my Ph.D. under his guidance. But sure enough, our shared love for multiprocessing, particularly our linked lines of research on the multicore algorithms for fast graph processing, brought us together. When I started with Julian, my research spanned fields, but all with a strong theoretical bent. Julian was pivotal to expanding my outlook to encompass both theory and experiment. The results of this thesis which span theory, experiment, and formal methods are a result of that expansion, which started in Julian's signature seminar *Algorithms Engineering*, and continued through our research together. On a more personal level, I am also grateful to Julian for his openness and caring. He has been there whenever I have needed his advice; whether it's something small or large, I could always count on Julian to answer my questions swiftly and thoughtfully. Julian was also incredibly supportive throughout my job search, from the advice he gave me on career options, to his careful critiques on my essays and talks, to the incredible freedom he gave me during that time period—allowing me full liberty to set aside research for months, so I could focus fully on my job search. Thank you Julian! I am fortunate to have had a Ph.D. advisor, who valued me both as a scientist and as a person.

I am incredibly grateful to my undergraduate advisor and Ph.D. committee member, Bob Tarjan. Bob introduced me to the joy and fulfillment of research while I was an undergraduate student at Princeton, and ever since then, he has been a close mentor, collaborator, and friend. Many important results in this thesis are joint work with Bob, and a large amount of the other work would not have happened had it not been for my studying under Bob. It is hard to overstate the impact that Bob has had on my life as a computer scientist.

I would also like to thank my final thesis committee member, Charles Leiserson. I have learned a lot about technical writing from him—lessons that have played a significant role in the writing of this thesis. I have also had long discussions with him about computing, and hope that someday we will get to collaborate on a research project.

I would like to thank Costis Daskalakis, my Master's advisor. Costis's clear, insightful, and

engaging lectures lured my interest towards machine learning, and our work together on machine learning, economics, and statistics—some of which appears in this thesis—played a foundational role in my growth as a researcher.

I want to extend a special thank you to Manjul Bhargava for introducing me to the beautiful intersections between mathematics and language through ancient Sanskrit texts. Manjul has been a close friend and mentor to me since my undergraduate years.

I am grateful to the several professors that welcomed me into their groups and were there for me whenever I went to them for advice. Thanks in particular to Srini Devadas, Nancy Lynch, Aleksander Madry, Silvio Micali, Vinod Vaikuntanathan, and Virginia Williams. Thanks also to the wonderful staff in the department for their friendship and support: Alicia Duarte, Janet Fischer, Debbie Goodwin, Joanne Hanley, Lynda Lynch, Patrice Macaluso, and Rebecca Yadegar.

I would like to thank the several teachers and professors who impacted me in the classroom, especially: my grade school teachers Thomas Cochran, Stephen Hackman, Jody Horan, John Kitzmiller, Amy Kono, Matt Prince, and Bill Murphy; my undergrad professors Emmanuel Abbe, Moses Charikar, Robert Gunning, and Bob Tarjan; and my grad school professors Costis Daskalakis, David Karger, Pablo Parrilo, Yury Polyanskiy, Julian Shun, and Ryan Williams.

I would like to thank all my collaborators and co-authors without whom this thesis would not have been possible: Marcos Aguilera, Naama Ben-David, Enric Boix, Yuval Dagan, Costis Daskalakis, Nishanth Dikkala, Ben Edelman, Lizzie Hernandez, Prasad Jayanti, Sucharita Jayanti, Anup Joshi, Srini Raghuraman, Siddhartha Sen, Julian Shun, Ankur Taly, Bob Tarjan, Nikhil Vyas, and Ugur Yavuz. It was a joy to work together—talking about problems over games, impromptu meetings and whiteboard jam-sessions, random 2am message threads with research breakthroughs, and sleepless nights before deadlines—and enjoy together—chatting about politics, playing ping pong or Goofspiel, or just chatting for hours in Stata.

Whether we were playing soccer, hitting up the Cheesecake Factory, eating late-night ice cream, or going rock climbing, some of my fondest memories from grad school have been with my friends. Thanks to *the Gaarus*, *the Board Game Beasts*, *the Zoom Lords*, *Theory Group* (including *Theory Lunch, Tea and Retreat*), *Hindu Students Council*, and *MIT Samskritam*. Thank you to: Aviv Adler, Josh Alman, Prabhanjan Ananth, Yamin Arefeen, Venkat Arun, Arjun Balasingam, Saketh Bhamidipati, Enric Boix, Justin Chen, Yuval Dagan, Mina Dalirrooyfard, Laxman Dhulipala, Nishanth Dikkala, Ben Edelman, Alireza Fallah, Max Fischelson, Rahul Ilango, Andrew Ilyas, Farnaz Jahanbakhsh, Samvit Jain, Ravi Jaishankar, Matthew Jin, Shreyas Joshi, Gautam Kamath, Ar-

nav Kejriwal, Jonathan Kenkle, Priya Kollipara, Niels Torben Kühlert, Allen Liu, Quanquan Liu, Alexander Madry, Frederik Mallmann, Dylan McKay, Slobodan Mitrovic, Vaikkunth Mugunthan, Akshay Narayan, Anand Natarajan, Vikram Nathan, Parimarjan Negi, Ravi Netravali, Parth Parihar, Soya Park, Peter Park, Riddhi Patel, Sarath Pattathil, Aniruddh Raghu, Govind Ramnarayan, Luke Schaeffer, Adam Sealfon, Abhin Shah, Sohil Shah, Jessica Shi, Sandeep Silwal, Nalini Singh, Sohini Sircar, Vibhaa Sivaraman, Jennifer Tang, Kapil Vaidya, Nikhil Vyas, and Nicole Wein.

Thanks to Parasara Duggirala for collaborating with me on *TeluguTeX*.

Lots of people have made foundational contributions to my intellectual, cultural, and spiritual growth. I fondly recall Senthil Periaswamy and Srdjan Petrovic for kindling my early interest in logical thinking through puzzles and chess. I offer my *pranāms* to Natarajan (Sethuraman) Uncle and Kamala (Natarajan) Aunty, and Sriram Annayya for passing on to me the traditional knowledge of the *Vedā*. Thanks also to *Hindu Students Council (HSC)*, the *Dartmouth Shanti* community and *Saraswati Mandiram*, *Samskrita Bharati*, *Vyoma Labs*, and *MIT Samskritam* for adding color and culture to my Ph.D. journey.

I have shared an apartment with my sister, Sucharita Jayanti, since the beginning of grad school, and also with my brother-in-law, Siddharth Agrawal, in my final year. During the COVID lock-down, my parents (Aparna and Prasad Jayanti), my sister and brother-in-law, and I got to spend over a year together at home in Hanover. Our time together—playing 2-v-3 basketball, writing joint research papers, and just enjoying family time—I will cherish forever.

Thanks to the United States Department of Defense for supporting my graduate studies through the *NDSEG Fellowship (National Defense Science and Engineering Graduate Fellowship)*.

I would also like to recall the efforts of my family in getting me to where I am today. If it were not for the endless hours that my parents spent teaching me since my childhood, I would not have gotten to MIT or this place in my life. My mother taught me to speak, read, and write Telugu before kindergarten and was my first *guru*. My father has been my single most significant educational mentor since my childhood. My sister is three years older than me, and I have benefited immensely from her caring advice and guidance as I navigated each stage in life: school, college, and beyond. My family's unique concoction of playing, learning, watching movies, and contemplating the world has been the biggest influence on both my educational and personal life. I am fortunate to be supported always by the unyielding love of my family—my parents, my wife, my sister and brother-in-law, my grandparents, my parents-in-law and brother-in-law, uncles, aunts, cousins and beyond. I recall my late grandfathers. *Āmpa*, my paternal grandfather Visveswara Sarma Jayanti,

was a deeply intellectual man, and I know he would have been very proud that I pursued a Ph.D. *Thāthatha*, my maternal grandfather Sundara Ramaiah Malladi, held a Ph.D. himself, and I know he dreamed of the same education for me. The sacrifices and encouragement of my family, including those that are no longer with us, were pivotal to my happiness and successes, and to the overall well-being of the family; this, I will always remember in my heart.

Finally, the single most significant event during my Ph.D. was my marriage. I dedicate this acknowledgement to my wife, Soundarya Jayanti.

सत्यमेव जयते नानृतं सत्येन पन्था विततो देवयानः ।

येनाक्रमन्त्यृषयो ह्याप्तकामा यत्र तत् सत्यस्य परमं निधानम् ॥

Truth alone triumphs; not falsehood. Truth, itself, unfurls the divine path by which sages conquer their desires and ascend to the abode of supreme Truth.

मुण्डकोपनिषद् ३.१.६

Mudakopanishad 3.1.6

Contents

Part I

Overview and Preliminaries

Chapter 1

Introduction

In this dissertation, I design efficient multicore algorithms, which exploit and showcase the power of modern multiprocessors by efficiently solving fundamental problems across disciplines. I identify four core design goals: simplicity, speed, scalability, and reliability, and I use these goals as guiding posts as I solve problems ranging the traditional areas of data structures, concurrency, formal methods, and machine learning. In my algorithms and analyses, I bring together techniques and ideas from several fields, such as fault-tolerance, asynchrony, and partial knowledge from distributed computing; communication, amortization, and randomization from algorithms; model specification, simulation, and invariance from machine-verification; and statistics, measure concentration, and Markov models from probability. My thesis is that, *by leveraging powerful technical tools across fields, we can design simple, fast, scalable, and reliable multiprocessor algorithms for fundamental problems across disciplines.*

1.1 Motivation

Speeding up computation enables us to develop novel technologies in all spheres of life, thus the design of efficient multicore algorithms is a key to current and future technological advancement. In the past, the exponential scale-down of transistors (Moore's Law [160]) under constant power density (Dennard Scaling [50]) permitted hardware manufacturers to drastically increase clock speeds, by about 30% per year [139], thereby enabling more computation per second in each new generation of the single-core microprocessor. In recent years however, fundamental physical barriers marked the end of Dennard Scaling, thereby resulting in an upper limit on clock speeds; in short, clock frequencies beyond around 5GHz melt chips [139]. This stop on increasing clock speeds in the

face of the world's ever growing computational needs sparked the *multicore revolution* [88, 139]. For the past two decades, hardware manufacturers have produced shared-memory *multi*processors with steadily increasing core-counts. Today, personal laptops and mobile phones are, almost universally, shipped with multiple processing cores, commodity machines run to over 200 cores [190], and cloud machines with up to 224-cores (448 logical processors with hyperthreading) and 24 terabytes of memory can even be rented on the *Amazon Cloud* [8]. To effectively transform the abundance of powerful multicores into efficient computation, we need fast multiprocess algorithms.

Multicore algorithms have traditionally come in two flavors: *asynchronous* and *synchronous*. Natively, a multicore machine is asynchronous, meaning that the many threads running on a machine can interleave arbitrarily. However, the difficulty inherent to designing asynchronous algorithms that behave correctly in all interleavings has spurred a lot of gainful research into synchronous parallelism, such as Fork-Join [167], PRAM (Parallel RAM) [101], and Cilk [26]. A lot of beautiful and efficient synchronous parallel algorithms have been discovered in the past decade [185]. However, synchronizing processes means that faster processes must often wait for slower processes to catch up before they can move on to their next task, thereby causing a *synchronization overhead* [73]. Furthermore, many multicore problems, such as those in *multi-tenant systems* [133]—in which each processor represents a different user—require efficient coordination between concurrent processes that may be solving fundamentally different problems, rather than working together to solve some single problem quickly; such problems of *concurrency* also need efficient algorithmic solutions. These considerations led to the formulation of the APRAM (Asynchronous Parallel RAM) model of computation [73], which models multiprocessors with their inherent asynchronicity and allows for the direct design of asynchronous algorithms for multiprocessors. In this thesis, I focus on such asynchronous algorithms for multiprocessors, thereby taking on the burden of designing algorithms that are robust to arbitrary interleavings, and simultaneously striving to squeeze out all the efficiency available in multiprocessors.

1.2 A Brief Illustration: Union-Find

Before expanding further on the design goals, challenges, and problems addressed, let me illustrate the flavor of this thesis, and how techniques across fields come together in addressing a specific problem I worked on: the union-find object. The collection of work on the concurrent union-find object presented in this thesis starts with a simple question: *can we use multiprocessors to speed up*

some of the most fundamental and ubiquitous graph computations, such as connected components, strongly connected components, and clustering? My research leverages recent developments in sequential data structures, such as the use of fixed randomized identifiers instead of changing ranks to ensure data structure efficiency [75]; recent developments in formal methods, such as the Temporal Language of Actions Proof System (TLAPS) [153]; and develops several new ideas on top of these to overcome fundamental barriers in distributed systems such as asynchrony and communications limits. The main algorithmic result is the first union-find object that provides "almost linear speed-up" as the number of processors available increases, which we prove is optimal by a mathematical information propagation bound that shows that fully linear speed-up cannot be achieved. In practice, the algorithm has led to improvements in the state-of-the-art in the speeds of model checking [23], spatial clustering [208], and computing connected and strongly connected components [51, 95, 23].

Background Social network, Web, and road graphs are humongous, and applications ranging from navigation to image segmentation rely on efficient graph analysis. A fundamental problem in graph analysis is computing connected components (CC). In fact, a recent study by Sahu et al. published in VLDB found that computing connected components is one of the most fundamental subroutines in graph processing and consequently the most popular graph computation in industry [177]. One of the fastest solutions to computing connected components is via the set union (a.k.a. union-find) object. In fact, the entire algorithm to compute the components of a graph is a single one-line loop that calls the object's "Unite" method on each pair of incident vertices. Consequently, a fast sequential set union algorithm implies a fast sequential CC algorithm, and a fast concurrent union-find algorithm yields a fast parallel CC algorithm.

The design and analysis of the fast "compressed forest" sequential set union data structure is one of the grandest accomplishments in data structure history. In fact, Tarjan's proof that m set union operation could be performed in just $O(m \cdot \alpha(n))$ time—where $\alpha(\cdot)$ is the extremely slow growing, almost constant, inverse-Ackermann function—is cited as a key contributor to his winning the Turing Award [129].

While concurrent set union was introduced in STOC 1991 [12], solutions were inefficient. The fastest concurrent algorithm previous to ours requires $O(m \cdot (\alpha(n)+p))$ work to perform m operations when p processes are run in parallel. While the inverse-Ackermann term is effectively constant in practice, the p that multiplies m indicates that the work per operation increases linearly in the

number of processors. This linear increase in work per operation, in effect means that using more processors does not reduce the time per operation.

I took on the concurrent set union problem with the aim of designing an algorithm that achieves a large speed-up.

Simplicity Collaborating with Prof. Bob Tarjan, I used randomization techniques to develop a family of simple concurrent union-find algorithms, in which each of the two operations, union and find, are implemented in under ten lines (see the pseudocode below, where the input parameters x and y are nodes and CAS is the atomic "compare-and-swap" instruction).

```
1: procedure FIND(x)
2:     v ← x
3:     while true do
4:         u ← v
5:         do twice
6:             v ← u.p
7:             w ← v.p
8:             CAS(u.p, v, w)
9:             if v = w return v
```

```
1: procedure UNITE(x, y)
2:     u ← x
3:     v ← y
4:     while true do
5:         if u = v then return
6:         else if u < v and CAS(u.p, u, v) then return
7:         else if u > v and CAS(v.p, v, u) then return
8:         u ← FIND(u)
9:         v ← FIND(v)
```

Our algorithms are robust to asynchronous scheduling, i.e., they remain correct even if the speeds of the various processors are vastly different and vary adversarially.

Scalability Using potential functions, we rigorously proved that the amortized work complexity of each data structure operation is merely $O\left(\alpha\left(n, \frac{m}{np}\right) + \log\left(\frac{np}{m} + 1\right)\right)$ in expectation, when m operations are done by p processes on a union-find instance of n nodes. Here, $\alpha(\cdot, \cdot)$ is the extremely slow growing two-term inverse-Ackermann function; in fact, α grows so slowly, that it is bounded by four in any conceivable application of the data structure [40]. Furthermore, we prove that the algorithm does at most $O(m \log n)$ work over any m operations with high probability. Thus, our algorithms are the first scalable algorithms for the problem, and since our algorithms' work complexity scales only *logarithmically* with the number of processors, they achieve *almost linear speed-up*. Furthermore, we show that our high probability result extends to the sequential random index compressed forest data structure, thereby resolving (affirmatively) the tree-height conjecture of Goel, Khanna, Larkin, and Tarjan [75].

Speed Our algorithms are lightning quick in practice. An MIT research group independently implemented hundreds of parallel algorithms for connected components and revealed that our algorithms are consistently the *fastest on CPUs* [51] and the *fastest on GPUs* [95]. As an illustration,

our algorithms were used to compute the components of the largest publicly available graph, the Hyperlink2012 graph of 128 billion edges, in just 8.2 seconds on a standard 72 core machine; this is 3.1× faster than the previous state-of-the-art in *any* computational setting [51]. Our algorithms have also improved the state-of-the-art in computing strongly connected components for model checking [23], and spatial clustering [208].

Reliability Our implementation is furnished with a machine-verified proof of correctness. Concurrent data structures are extremely difficult to prove correct, as it must be shown that every run of an algorithm using the data structure satisfies a consistency condition known as *linearizability*. Linearizability has been the long standing gold standard for consistency in multiprocess data structures, since it ensures that operations appear to take place instantaneously even in the face of tremendous concurrency and asynchrony. Yet, proving linearizability is notoriously difficult. Proofs of linearizability can be long and intricate, hard to produce, and extremely time consuming even to verify. To address this issue, my collaborator Prof. Prasad Jayanti and I designed a *universal*, *sound*, and *complete* proof method for producing machine-verifiable proofs of linearizability. Universality means that our method works for any object type; soundness means that an algorithm can be proved correct by our method only if it is linearizable; and completeness means that any linearizable implementation can be proved so using our method. Using this technique, I gave a proof of correctness, for our union-find implementation that is human readable, and machine-verified by TLAPS (the temporal logic of actions proof system) [153].

Further Investigation Inspired by the impact of the algorithms, I questioned whether I could design an even faster union-find algorithm with *fully linear speed-up*—the algorithmic dream. This line of inquiry led me to show some sharp impossibility results. I demonstrated that several concurrent objects such as union-find, queues, and stacks are each powerful enough to solve *generalized wake-up*—a fundamental communication task that I defined and proved to be computationally hard. In particular, I showed that fully linear speed-up is not achievable for any of these data structures. With a finer mathematical analysis, I proved that our union-find object's work complexity is asymptotically optimal for the class of *symmetric algorithms*, which captures the complexities of all known concurrent union-find algorithms, by deriving a matching $\Omega\left(\alpha\left(n, \frac{m}{np}\right) + \log\left(\frac{np}{m} + 1\right)\right)$ lower bound for all such algorithms. All my lower bounds are unconditional and cannot be breached by the use of randomization or amortization.

In the remainder of this chapter, I will abstract away from the union-find example, and discuss the four design goals, their interplay, and the challenges inherent to achieving them. I describe the problems studied in this thesis, and my main contributions in the next chapter, and wrap-up the introductory part in the subsequent chapter on preliminary notation and definitions.

1.3 Algorithmic Design Goals

In this dissertation, I identify four algorithmic goals to facilitate in the design of efficient multicore algorithms, namely, *speed*, *scalability*, *reliability*, and *simplicity*. The individual goals are well established, in the sense that most, if not all, computer scientists will immediately agree to their importance. However, it can be difficult to achieve them *simultaneously*. Designing multicore algorithms that simultaneously achieve all these goals is the primary pursuit of this thesis.

Speed Finishing a computation in as small an amount of time as possible, is a fundamental goal of the algorithmist. Tangibly, lowering running times can correspond to faster response times and refresh rates in end-user applications, lower rental costs when using cloud machines—such as AWS (Amazon Web Services) machines, for which cost is proportional to rental time—and greater energy efficiency for a given computation. In other words, faster amounts to cheaper, greener, and more user-friendly. The fundamental algorithms and data structures that I discuss in this thesis—such as arrays, union-find, hash tables, mutual exclusion locks, and Gibbs Sampling—are used ubiquitously. Thus, speed-ups to these computations percolate to all spheres of software. Traditionally, the "speed" of an algorithm, has been measured both mathematically, as *time complexity* (a.k.a. *work complexity*), and practically, by measuring the *wall clock time* an algorithm takes to compute on inputs of interest on actual modern hardware. In this thesis, I emphasize the importance of *both* theoretical and practical speed.

Scalability As the world is looking to process larger and larger data sets and, simultaneously, the multicore revolution is rapidly driving up core counts, it is important to design algorithms whose theoretical complexity scales well in both data size and core-count; in this dissertation, we call such algorithms *scalable*. The importance of theoretical scalability is well illustrated by history. The abstract of Bob Tarjan's celebrated Ph.D. thesis, published in 1971, reads

" An efficient algorithm is presented for determining whether a graph G can be embedded in the plane. Depth-first search, or backtracking, is the most important of the techniques used by the algorithm. If G has V vertices, the

algorithm requires $O(V)$ space and $O(V)$ time when implemented on a random access computer. An implementation on the Stanford IBM 36067 successfully analyzed graphs with as many as 900 vertices in less than 12 seconds." [195]

The abstract makes two statements about the algorithm's efficiency: one about its practical speed on a modern machine, and one about its scalability with respect to the input size. Just 50 years after this publication, we live in a world where a single mobile phone processsor can do three billion operations per second, rendering the practical speed statement about seven orders of magnitude out of date. Its scalability guarantee however, is what has immortalized the linear time planarity-testing algorithm, which has remained a fast solution throughout generations of computational hardware.

Sharply rising core-counts are the hallmark of the multicore revolution. Still in its infancy, this revolution has already driven core-counts up to the hundreds (incidentally, similar to the graph input size Tarjan mentioned fifty years ago). Simultaneously, we continue to live in the age of big data. So, lasting algorithms in the multicore era must remain efficient as input sizes get larger and exploit the parallelism presented by growing core-counts. That is, they must exhibit scalability both with respect to input size and core-count.

Reliability Mutiprocessors are our technological present and future, however designing correct algorithms for multiprocessors is notoriously hard. While a deterministic single-process algorithm has exactly one possible run, due to asynchrony, even a t step concurrent algorithm with just two processes has 2^t possible runs depending on how the steps of the processes interleave. When we consider algorithms with an infinite horizon, even a deterministic concurrent algorithm has uncountably many possible infinite runs. Designing algorithms that are correct in all of these executions is a grueling task, and programmers often fail to account for some of these executions, leading to subtle and dangerous bugs, known as *races*. Races are pernicious, since they can easily be missed in testing but have harsh consequences when deployed in practice. For example:

- **Mars Rover:** a priority inversion bug in its concurrent code crashed the Pathfinder Rover days after its deployment on Mars and jeopardized the entire multi-million dollar NASA space mission [124, 123].
- **Northeast Blackout of 2003:** a race in the power grid's energy management system stalled the alarm system for an hour, by which time it was too late to stop a cascading electrical outage that affected an estimated 55 million people in eight U.S. states and the province of Ontario, Canada [170].

- **Therac-25:** the software of the radiation therapy machine, Therac-25, suffered from races that caused it to administer radiation doses that were over a hundred times as potent as the intended dose, which caused the deaths of at least three people and several more injuries [140, 142, 139].

These illustrations show just how fatal the consequences can be, if critical multiprocessor code is not correct. But which pieces of code should we consider critical when developing algorithms? Two foremost guiding principles in the design of software technology are *modularity* and *top-down* design. Together, these principles state that larger applications should be broken down into smaller well-specified components, i.e., methods, data structures, and algorithms, and that these modular components should be developed independently and efficiently, so they can be called and used in several high-level applications. A strength of this prevalent design strategy is that each core modular component can be built and developed in just one place, and the same well-built component can be trusted and used freely in a range of applications today and even the unforeseen applications of tomorrow. The flip-side of this advantage is a corresponding failure mode, which I term *error proliferation*. Namely, if a core component such as a data structure or algorithm has an uncaught error, it runs the risk of being used in innumerable applications, and ultimately crashing critical systems. Even if the component was developed with a low-risk application in mind, the principles of software design make it very easy for the same component to later be integrated into a critical application. Thus, I argue that, for all intents and purposes, *all fundamental data structures and algorithms should be considered critical*, and rigorous mathematical proofs of correctness are indispensible as we develop multicore algorithms.

Guarantees of efficiency and other properties of algorithms are often also pivotal to end-user applications. Thus, in this thesis, I emphasize reliability to constitute rigorous demonstrations of correctness, efficiency, and any other relevant properties.

Simplicity A principle that is synergistic with all of the goals we have just seen is simplicity. Simple algorithms, which use few variables and few lines of code to achieve their specification, have several advantages.

- First, simple algorithms are easier to reason about, making the algorithmist's job of clearly specifying and proving-correct the algorithm easier. Simpler and more straight-forward reasoning in proofs of correctness and efficiency, in turn, makes proof verification easier for reviewers and end-users. Thus, simplicity aids reliability.

- Second, simple algorithms are easier for a software engineer to understand and integrate correctly into production code. A proved algorithm is of little significance, if the final code in production systems does not replicate the algorithm faithfully. Thus, keeping the pipeline between algorithm design and the end-user in mind, we note that simplicity aids deployability.

- Third, a simple algorithm generally has lower constant factors and overheads. Thus, simplicity aids speed and scalability.

Overcoming Challenges While all four design goals are highly desirable, they can be hard to achieve together: famously, many beautiful, theoretically scalable algorithms are not fast in practice (such as fast matrix multiplication [6, 39] and near-linear max-flows [33, 130]); likewise, many simple, and practically fast algorithms are not reliable (perhaps most notoriously, neural networks [14, 203]). In this dissertation however, I work hard to design multicore solutions—like the concurrent union-find object of the previous section—that satisfy all four design criteria simultaneously. I end this section with a couple more examples.

- Reliability is particularly hard to achieve in multiprocess algorithms due to asynchrony. Perhaps striking, the rigorous mathematical proofs backing a sequential algorithm can break entirely even when the algorithms are adapted, just slightly, for multiple processors. In Chapter 12 of this thesis, I study a simple, fast, and scalable asynchronous Gibbs Sampling algorithm born out of Machine Learning practice [187]. With a thirst for speed, the practitioners simply replace the sequential **for**-loop of the standard Gibbs sampling algorithm with a **parallel for** to allow all the processors in a multicore to achieve maximum throughput in parallel. However, this technique immediately gives rise to a heap of race conditions that invalidate all proofs about the sampling procedure's convergence, rendering it *de facto* unreliable. The goal of the practitioners is to use Gibbs sampling to get samples $x_1, \ldots, x_n$ from a specified distribution D, and use these samples to estimate statistics of D. Our analysis of the resultant multicore algorithm reveals that the samples computed by it may not be accurate, i.e., the parallel-for introduces a bias. However, leveraging recently proved measure concentration bounds for high temperature graphical models and novel filtration based coupling arguments, we derive mathematical bounds on the bias that reveal that the algorithm's samples can be used to reliably compute approximately correct statistics in many applications of interest (see Chapter 12 for details).

- Several problems have solutions in the literature that offer a trade-off between the various design criteria. For instance, this was the case for the *abortable mutual exclusion* problem, which is to design a multicore lock that additionally allows for a process to *abort*, i.e. abandon the lock and switch to another task, in real-time (see Chapter 5 for details). Abortable locks are designed for two different types of machines called Cache-coherent (CC) and Distributed Shared Memory (DSM); their efficiency is measured in, so called, Remote Memory Reference (RMR) complexity. The literature offered three beautiful, yet incomparable solutions for the problem: Jayanti's "tree lock" [104], Lee's "procession lock", and Giakkoupis and Woelfel's "array lock" [72]. The tree lock is simple and offered the most robust scalability guarantee of $O(\log n)$ RMR complexity, but just for DSM machines. The procession lock was practically fast and had simple code, but its analysis offered only a weak $O(n)$ RMR complexity bound, and that too just for CC machines (we have subsequently improved this bound in our work, see Lee Alg 2 in Figure 5.1 in Chapter 5). The array lock has an impressive amortized $O(1)$ RMR complexity guarantee for CC machines, but the guarantee is only against oblivious schedules. Furthermore, its high code complexity and large space complexity make it hard to implement practically. I sought out this problem to try and design a single, simple and scalable solution that can be reliably deployed on both types of machine and is robust to even adversarial schedules. In Chapter 5, I present our *abortable queue lock*. This lock is fast, simple to code (see Figure 8 in Chapter 5), guarantees an amortized $O(1)$ RMR complexity for *both* CC and DSM machines, and additionaly, is the first lock to guarantee aborts in just six lines of code. All our claims are proved via rigorous mathematics: correctness is proved via invariants, liveness via distance measures, and efficiency via potential functions. In addition, I have modeled checked our implementation for CC machines via the TLC [59] model checker.

In this thesis, I design several such simple, fast, scalable, and reliable multiprocessor algorithms.

1.4 Broadening Participation in STEM

Contributions Along with the several technical contributions to multiprocessor algorithms, a significant, and very different contribution of this thesis is towards broadening participation in STEM (Science, Technology, Engineering, and Mathematics). In particular, I contribute, to the best of my knowledge, the first computer science research paper (presented in Chapter 10) in the Telugu language. Telugu is the thirteenth largest language in the world with over 80 million

native speakers [189], and the fastest growing language in the United States according to the World Economic Forum and US Census Data [21, 199]. At the beginning of the Telugu chapter, I present an additional abstract for the work in Sanskrit. Sanskrit is the principal source of technical vocabulary across Indian languages and many other languages around the world, thereby making this abstract accessible to an even larger group of people. The technical research contributions of this chapter, which pertain to concurrent data structure lower bounds, are also reproduced with some extensions in English (Chapter 9), making them equally accessible to the large community of English language researchers. I hope that this initiative can lay the foundation for many more works that make scientific discovery more available to people of various languages.

Motivation Spreading the joy of learning, especially scientific and technical learning, has been a passion of mine since my childhood. As a child, I was touched to hear of my father's lived-experience of how his education and hard work pulled our family out of the poverty and hopelessness of 1960s India and brought us to the USA. Inspired by our family's journey and many more like it, I recognized that a good education paves the way to a prosperous and joyful future, especially for those who come from a socially or economically disadvantaged background. Unfortunately, as I learned first-hand in my time volunteering as a teacher of mathematics, robotics, and English in cities and towns in New Hampshire, New Jersey and some small villages in India, a good education is not easy to come by. Many children around the world go to underfunded schools, lack committed teachers, have unfortunate home environments that make studying difficult or impossible, or struggle to learn in a second language since their native language lacks adequate educational resources. I am passionate about creating learning opportunities for students at all levels of education and around the world, and am keen on encouraging students of diverse backgrounds and identities in varied circumstances to pursue the joys of learning, science, and even furthering science through their curiosity and discovery.

In spite of several commendable large scale efforts to promote scientific and technological education throughout the world, language remains a great barrier to scientific learning. Enabling progress through scientific education has been a core tenet of many prominent institutions. Notably in the United States, we have the National Science Foundation, which "is committed to expanding the opportunities in STEM to people of all racial, ethnic, geographic and socioeconomic backgrounds, sexual orientations, gender identities and to persons with disabilities." [29] In India, the government recently undertook a gargantuan initiative to write a new educational policy from

pre-K to university to better serve a billion people. One of their major findings was that education in general, and science in particular, are easier for people to grasp in their mother tongues. Yet, several hundreds of millions of people in India alone are currently unable to receive quality education in their mother tongues due to lack of educational resources [82]. The new educational policy observes that there are "students going to school to classes that are being conducted in a language that they do not understand, causing them to fall behind before they even start learning," and that "textbooks (especially science textbooks) written in India's vernaculars at the current time are generally not nearly of the same quality as those written in English. It is important that local languages, including tribal languages, are respected and that excellent textbooks are developed in local languages, when possible, and outstanding teachers are deployed to teach in these languages" [126] (P4.5.0). In short, the unavailability and under-availability of science in local languages is disadvantaging hundreds of millions of children.

Impact My article on multiprocessor algorithms in Telugu, which, to my knowledge, is the first computer science research paper in the language, aims to serve as a starting point to ameliorating this situation for speakers of Telugu and other languages alike. After publishing the original article on $ar\chi iv$ [215, 120], I was touched to receive several communications from researchers and scientists from across institutions, including Princeton, CMU, University of Chicago, Google Research, and IIT-Delhi, seeking to learn more about this initiative and expressing interest in both translating some of their work into other languages and creating other resources for scientific learning in several world languages. In the thesis, I include an abstract in Sanskrit—the language in which the technical terms of most Indian languages and several other world languages are derived (like Latin and Greek for English and Romance languages). I hope this inclusion captures the imagination of even more people, and spurs science-availability in even more language communities. (I share some more thoughts and concrete proposals in the final chapter.)

Knowledge is universal, brings joy, opens doors to new opportunities, and has the power to enlighten and bring people of diverse backgrounds closer together in pursuit of a better world. My scientific learnings and discoveries, some of which are presented in this dissertation, have been a great joy to me, have brought me in contact with great minds around the world, and have served as a conduit to reveal some of the infinite possibilities this universe has to offer. I hope that at least some of my work can open up such a gateway for a few more people in the world.

Chapter 2

Contributions

In this thesis, I design and mathematically analyze simple, fast, scalable and reliable multiprocess algorithms. I also develop rigorous proof methods and machine verification techniques to prove the correctness of subtle concurrent algorithms. In the remainder of this section, I describe the specific contributions of this thesis.

2.1 Summary of Contributions

1. Part II focuses on efficient mutual exclusion locks, including *stardard mutex locks* (Chapter 4), *abortable locks* for real-time systems (Chapter 5), and *recoverable locks* for modern persistent memory such as Intel Optane (Chapter 6).

2. Part III focuses on multiprocess data structures that are lock-free—i.e., do not make use of mutual exclusion locks—and linearizable—i.e., appear to perform each data structure operation atomically.

   (a) In Chapter 7, we design and analyze concurrent multiprocessor algorithms for the *set-union* (i.e., *union-find*) data structure. Our data structures have led to significant practical speed-ups in computing connected components, model checking, and clustering.

   (b) In Chapter 8, we design and analyze the first algorithms for concurrent *fast arrays* and *fast generalized arrays*. Fast arrays support constant time initialization (of the whole array!), and constant time reads and writes. Fast generalized arrays additionally support, in constant time, every atomic operation offered by the underlying hardware, such as compare-and-swap (CAS), fetch-and-store (FAS/swap), etc. As an example application

of these arrays, we design a concurrent fixed-size hash table that supports expected constant time initialization, insertion, and search.

(c) In Chapter 9, I define a new information propagation problem called *generalized wake-up*, show its hardness, and through reductions show that many common data structures such as stacks, queues, priority queues, and union-find objects cannot be implemented for multiple processes without a concurrency overheard, i.e., a work complexity that grows with the number of processes, even in the amortized sense.

(d) Chapter 10 (truly, అధ్యాయము 10) is clearly special, as it is written entirely in Telugu. The chapter is actually an authentic reproduction of my original article on generalized wake-up (జాగృతిపరిష్కారం, *Jāgrtiparişkāram*) lower bounds. Due to the prevalence of English readers in the computer science research community, I translated the work into English, and composed Chapter 9.

3. Part IV has a single chapter, Chapter 11, in which we derive a technique for writing machine verifiable proofs of linearizability and strong linearizability of concurrent data structures. We have employed our techniques using the TLAPS (Temporal Language of Actions Proof System) to produce machine-verified proofs of the linearizability and strong linearizability of our union-find data structures.

4. Part V has a single chapter, Chapter 12, in which we discuss a technique called HOGWILD! Gibbs Sampling, which uses an asynchronous multiprocessor to improve the speed of sampling certain high dimensional probability distributions that are of great interest in Machine Learning applications. The statistical analysis we develop justifies the use of HOGWILD! Gibbs sampling, a technique born out of practice that uses asynchronous multiprocessors to speed up the simulation of Gibbs sampling, in spite of its severe race conditions.

The chapters of this thesis are based on several of my co-authored publications. I list these publications below, classified as *Journal Articles*, *Conference Publications*, and *Articles to be Published*.

Journal Articles

1. Siddhartha Jayanti and Robert Tarjan. Concurrent Disjoint Set Union. In *Distributed Computing* 2021. [119]

2. Prasad Jayanti and Siddhartha Jayanti. Deterministic Constant-Amortized-RMR Abortable Mutex for CC and DSM. In *ACM Transactions on Parallel Computing* 2021. [107]

Conference Publications

1. Siddhartha Jayanti and Julian Shun. *Fast Arrays: Atomic Arrays with Constant Time Initialization*. International Symposium on Distributed Computing (DISC) 2021. [116]

2. Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. *Towards an Ideal Queue Lock*. International Conference on Distributed Computing and Networking (ICDCN) 2020. [108]

   Invited to the spotlight session of the conference

3. Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. *A Recoverable Mutex Algorithm with Sub-logarithmic RMR on Both CC and DSM*. ACM Symposium on Principles of Distributed Computing (PODC) 2019. [110]

   Invited to a special issue of the journal *Distributed Computing*

4. Prasad Jayanti and Siddhartha Jayanti. *Constant Amortized RMR Complexity Deterministic Abortable Mutual Exclusion Algorithm for CC and DSM Models*. ACM Symposium on Principles of Distributed Computing (PODC) 2019. [113]

5. Siddhartha Jayanti, Robert Tarjan, and Enric Boix-Adserà. *Randomized Concurrent Set Union and Generalized Wake-Up*. ACM Symposium on Principles of Distributed Computing (PODC) 2019. [117]

   Invited to a special issue of the journal *Distributed Computing*

6. Constantinos Daskalakis, Nishanth Dikkala, and Siddhartha Jayanti. *HOGWILD!-Gibbs can be PanAccurate*. Neural Information Processing Systems (NeurIPS) 2018. [44]

7. Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. *Optimal Recoverable Mutual Exclusion using only FASAS*. International Conference on Networked Systems (NETYS) 2018. [111]

8. Siddhartha Jayanti and Robert Tarjan. *A Randomized Concurrent Algorithm for Disjoint Set Union*. ACM Symposium on Principles of Distributed Computing (PODC) 2016. [118]

   Invited to a special issue of the journal *Distributed Computing*

Articles to be Published

1. సిద్ధార్థ విశ్వేశ్వర జయంతి. సామాన్య జాగృతిపరిష్కారం. [215]

2. Siddhartha Jayanti and Prasad Jayanti. *A Univeral, Sound, and Complete Technique to Facilitate Machine-Verifiable Proofs of Linearizability.*

For completeness, I also list the other co-authored and single-author articles I have written as a Ph.D. student.

Articles not part of this Thesis

1. Enric Boix-Adserà, Benjamin L. Edelman, and Siddhartha Jayanti. The Multiplayer Colonel Blotto Game. In *Games and Economic Behavior* 2021. [28]

2. Siddhartha Jayanti, Srinivasan Raghuraman, and Nikhil Vyas. *Efficient Constructions for Almost-everywhere Secure Computation.* International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 2020. [115]

3. Enric Boix-Adserà, Benjamin L. Edelman, and Siddhartha Jayanti. *The Multiplayer Colonel Blotto Game.* ACM Conference on Economics and Computation (EC) 2020. [27]

4. Yuval Dagan, Constantinos Daskalakis, Nishanth Dikkala, and Siddhartha Jayanti. *Learning from Weakly Dependent data under Dobrushin's Condition.* Conference on Learning Theory (COLT) 2019. [43]

5. Siddhartha Jayanti. *Nash Equilibria of The Multiplayer Colonel Blotto Game on Arbitrary Measure Spaces.* [114]

In the succeeding sections, I describe the contributions of each part of the thesis in more detail.

2.2 Contributions to Mutual Exclusion Locks

Key to most shared multiprocess algorithms are *mutual exclusion (mutex) locks*, which allow a process to gain exclusive access to a shared resource. Such exclusive access control can be pivotal to ensuring consistency of data streams, such as printers, in the face of asynchronous concurrency. Mutex locks also facilitate the use of arbitrary sequential data structures by concurrent processes by allowing at most one process, i.e. the owner of the lock, to access the data structure at a given

time. Since sequential data structures are easier to design in practice than specialized concurrent ones, locks are ubiquitous in practice, and the design of efficient mutex locks is a cornerstone of efficient concurrent computation.

In Part II of this thesis, we design several efficient and performant mutual exclusion locks for standard, real-time, and persistent shared-memory multiprocessor systems, as described in the remainder of the section.

A new queue lock for standard mutual exclusion

My contributions to standard mutual exclusion locks are presented in Chapter 4.

The MCS lock (Mellor-Crummey Scott lock) was the first mutual exclusion lock to support an arbitrary number of processes with unknown identities such that each process can acquire and release the lock in a constant number of RMRs (Remote Memory References)[1] on both Cache-Coherent[2] and Distributed Shared Memory[3] multiprocessors. Additionally, this lock is *stateless*, i.e., a process does not have to remember any information after it releases the lock, and it has *adaptive* space complexity, i.e., the size of shared and local memory required by the algorithm depends only on the number of active processes. The MCS algorithm, however, has two shortcomings: its Exit section (i.e., unlock code) is not bounded and it requires more than one special instruction, namely, *Fetch-and-Store* and *Compare-and-Swap*. Many MCS-style algorithms were subsequently designed to overcome these shortcomings, but to the best of our knowledge they either lack some of the other desirable properties of the MCS lock or introduce a new shortcoming. In this chapter, we present a new MCS-style algorithm that has all of the desirable properties and no ostensible shortcoming. We also provide an invariant-based proof of correctness. As far as we know, ours is the first among the MCS-style algorithms to be accompanied by a rigorous invariant-based proof, which is important given how popular these locks are in practice. To realize a bounded Exit section, all prior MCS-style algorithms use either the "node-switching" or the "node-toggling" strategy. Our work unifies these two strategies: we present a single algorithm which, when appropriately instantiated, yields both a node-switching and a node-toggling algorithm. Moreover, the two algorithms so derived are the simplest in their respective classes among all known MCS-style algorithms. Table 2.1 compares our

[1] RMR complexity is the standard metric for measuring the efficiency of mutual exclusion locks. In essence, it counts the number of instructions performed on memory that is expensive to access.

[2] In the Cache-Coherent model, cached variables can be read at no cost, but all other instructions cost a single unit.

[3] In the Distributed Shared Memory model, shared memory is partitioned amongst the processors. instructions performed on memory in a processes's own part are at no cost, and instructions performed on shared memory in any other part cost a single unit.

algorithm with previous works.

Table 2.1: Comparison of Existing Mutual Exclusion Algorithms

	$O(1)$ **RMR for DSM**	**Bounded Exit**	**FCFS**	**Stateless**	**Space Frugal**	**Supports Arbitrary Set of Processes**	**Number Special Instruction**	**Other Drawbacks**
Anderson [13]	No				No	No	1	
Graunke, Thakkar [83]	No						1	
MCS [152]		No					2	
Craig [41]							1	not memory safe
Magnusson et al. [147]	No						1	
Rhee [176]		No		No	No		1	
Lee Alg 1 [138]				No	No	No	1	uses $\Theta(N^2)$ space
Lee Alg 2 [138]		No		No	No	No	1	
Anderson, Kim [10]			No		No	No	1	
Dvir et al. Alg 1 [56]				No	No		2	
Dvir et al. Alg 2 [56]							2	
This thesis (main)							1	
This thesis (toggling)				No	No		1	
This thesis (switching)							1	

The fast abortable queue lock

My contributions to abortable mutual exclusion locks are presented in Chapter 4.

The Abortable mutual exclusion problem, proposed by Scott and Scherer in response to the needs in real time systems and databases, is a variant of mutual exclusion that allows processes to abort from their attempt to acquire the lock. Worst-case constant remote memory reference (RMR) algorithms for mutual exclusion using hardware instructions such as Fetch-and-Add or Fetch-and-Store have long existed for both Cache Coherent (CC) and Distributed Shared Memory (DSM) multiprocessors, but no such algorithms are known for abortable mutual exclusion. Even relaxing the worst-case requirement to amortized, algorithms are only known for the CC model.

In this chapter, we improve this state-of-the-art by designing a deterministic algorithm that uses Fetch-and-Store (FAS) to achieve amortized $O(1)$ RMR in both the CC and DSM models. Our algorithm supports Fast Abort (a process aborts within six steps of receiving the abort signal), and has the following additional desirable properties: it supports an arbitrary number of processes of arbitrary names, requires only $O(1)$ space per process, and satisfies a novel fairness condition that we call "Airline FCFS". Our algorithm is short with fewer than a dozen lines of code. Table 2.2 compares our algorithm with previous abortable locks.

The sublogarithmic recoverable lock

My contributions to recoverable mutual exclusion locks are presented in Chapter 6.

Algorithm	Primitive	RMRs	WC / Amrt	Det.	Space	DSM	Fairness	Fast Abort
Scott et al. [180]	FAS, CAS	∞	WC	✓	∞	✓		
Scott CLH-NB [179]	FAS, CAS	∞	WC	✓	∞	✓		
Scott MCS-NB [179]	FAS, CAS	∞	WC	✓	∞	✓		
Jayanti [104]	CAS	$\Theta(\log n)$	WC	✓	$\Theta(n)$	✓	FCFS	
Lee Alg 1 [138]	None	$\Theta(\log n)$	WC	✓	$\Theta(n \log n)$	✓		
Lee Alg 2 [138]	FAS, CAS	$\Theta(n)/\Theta(1)$	WC/Amrt	✓	$\Theta(n)$		AFCFS	
Lee Alg 3 [138]	FAS	$\Theta(n^2)$	WC	✓	∞		FCFS	
Lee Alg 4 [138, 137]	FAS	$\Theta(n^2)$	WC	✓	$\Theta(n^2)$		FCFS	
Woelfel et al. [169]	CAS	$O(\frac{\log n}{\log \log n})$	WC		$\Theta(n)$			
Giakkoupis et al. [72]	CAS	$\Theta(1)$	Amrt		∞			
Alon et al. [7]	F&A, CAS	$O(\frac{\log n}{\log \log n})$	WC	✓	$O(n^2)$			
This thesis (CC)	FAS	$\Theta(1)$	Amrt	✓	$\Theta(n)$		AFCFS	✓
This thesis (DSM & CC)	FAS	$\Theta(1)$	Amrt	✓	$\Theta(n)$	✓	AFCFS	✓

Table 2.2: Comparison with existing abortable locks. The columns describe: RMR complexity; whether the complexity is worst case (WC) or amortized (Amrt); whether the algorithm is Deterministic (Det.) or randomized; space complexity; whether the RMR bound holds for the DSM model; what fairness condition (if any) the algorithm satisfies; and whether the algorithm supports Fast Abort.

In light of recent advances in non-volatile main memory technology, Golab and Ramaraju reformulated the traditional mutex problem into the novel *Recoverable Mutual Exclusion* (RME) problem. In the best known solution for RME, due to Golab and Hendler from PODC 2017, a process incurs at most $O\big(\frac{\log n}{\log \log n}\big)$ remote memory references (RMRs) per passage, where a passage is an interval from when a process enters the Try section to when it subsequently returns to Remainder. Their algorithm, however, guarantees this bound only for cache-coherent (CC) multiprocessors, leaving open the question of whether a similar bound is possible for distributed shared memory (DSM) multiprocessors.

We answer this question affirmatively by designing an algorithm that satisfies the same complexity bound as Golab and Hendler's for both CC and DSM multiprocessors. Our algorithm has some additional advantages over Golab and Hendler's: (i) its Exit section is wait-free, (ii) it uses only the fetch-and-store (i.e., swap) instruction, and (iii) on a CC machine our algorithm needs each process to have a cache of only $O(1)$ words, while their algorithm needs $O(n)$ words.

A recent result of Chan and Woelfel, shows that our algorithm's RMR complexity is optimal (up to constant factors) for both the CC and DSM machine models [32].

2.3 Contributions to Lock-Free Data Structures

While locks facilitate the concurrent use of data structures, they do not allow parallel access. Thus, distributed computing researchers design specialized data structures that guarantee consistency

(linearizability), while guaranteeing that all interested processes can perform operations on the data structure simultaneously (wait-freedom). The design of correct linearizable wait-free data structures is notoriously difficult due to asynchrony between processes, and the design of efficient such data structures is harder yet. However, scalable wait-free concurrent data structures and algorithms promise great computational speed-ups in this age of multiprocessors.

In Part III, we design and analyze efficient lock-free data structures for the union-find object, fast arrays, and fixed-size hash tables. We also prove lower bounds on the efficiency of several data structures, such as: stacks, queues, union-find objects, priority queues, and counters.

Concurrent disjoint set union algorithms

My algorithmic work on concurrent disjoint set union is presented in Chapter 7.

The *disjoint set union* or *union-find* problem is to maintain a collection of disjoint sets, each containing a unique element called its *leader*, under two operations:

FIND(x): return the leader of the set containing element x.

UNITE(x, y): if elements x and y are in different sets, unite these sets into a single set and designate some element in the new set to be its leader; otherwise, do nothing.

Each initial set is a singleton, whose leader is its only element. Note that the implementation is free to choose the leader of each new set produced by a unite.

In this chapter, we develop fast linearizable wait-free concurrent algorithms for union-find. In particular, we design a randomized concurrent implementation of *unite* using the read, write, and CAS instructions. We also give two concurrent implementations of path splitting, *one-try* and *two-try splitting*. The former is simpler, but we are able to prove slightly better bounds for the latter, bounds that we think are tight for the problem. We prove that any of our linking methods in combination with one-try splitting does set union in $O\left(m \cdot \left(\log\left(\frac{np^2}{m}+1\right)+\alpha\left(n,\frac{m}{np^2}\right)\right)\right)$ work, where m is the total number of operations across all processes, n is the number of nodes, and p is the number of processes. Our algorithm with two-try splitting does $O\left(m \cdot \left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n,\frac{m}{np}\right)\right)\right)$ expected work. Each set operation takes $O(\log n)$ steps[4] with (very) high probability. The $O(\log n)$ step bound per operation holds even without path splitting; without splitting, the high probability total work bound is $O(m \log n)$. The work and step bounds for randomized linking by rank hold even for an adversarial scheduler, provided that scheduling is based only on information sent to the

[4]Each primitive instruction (i.e., read, write, or another synchronization instruction like CAS) executed by a process is considered a *step*.

scheduler, or we allow a form of CAS that writes a random bit.

Our work is theoretical, but Alistrah et al. [3], Dhulipala et al. [51], Hong et al. [95] have implemented some of our algorithms on CPUs and GPUs and experimented with them. On many realistic data sets, our algorithms run as fast or faster than all others [51, 95].

Concurrent Fast Arrays

My work on concurrent fast arrays is presented in Chapter 8.

Arrays are the most fundamental data structure in computer science. Semantically, an *array* of length m is an object that supports the following interface:

- INITIALIZE(m, f): return an array $\mathcal{O}$ initialized to $\mathcal{O}[i] = f(i)$ for each $i \in [m]$.[5]
- $\mathcal{O}$.READ(i): return $\mathcal{O}[i]$, if $i \in [m]$.
- $\mathcal{O}$.WRITE(i, v): update $\mathcal{O}[i]$'s value to v, if $i \in [m]$.

Here, INITIALIZE() is the *constructor* method that creates the object, and READ() and WRITE() are the regular operations an array supports. Ordinarily, initialization is achieved by allocating an array of length m and looping through to initialize its entries, while reads and writes simply use the hardware load and store instructions. This standard implementation achieves a space complexity of $O(m)$, and time complexities of $O(m)$ for initialization and $O(1)$ for reads and writes. A *fast array* is an array that support all three operations—READ(), WRITE(), and *even* INITIALIZE()—in just $O(1)$ worst-case time. Perhaps surprisingly, sequential fast array implementations have been known for decades, and have been used in implementations of *van Emde Boas trees* [40, 161], hash tables [127, 131], and adjacency matrices for sparse graphs; but, to the best of our knowledge, concurrent implementations did not previously exist.

In this thesis, we propose and design algorithms for two variants of concurrent fast arrays:

- Fast Array: This is an implementation of an array which supports the operations—INITIALIZE(m, f), READ(i), and WRITE(i, v)—and satisfies two conditions. First, each operation is *linearizable*, i.e., it appears to take effect at some instant between its invocation and response [93]. Second, each operation is not only *wait-free* [89], but the process that executes the operation completes it in a constant number of its steps. The first condition ensures *atomicity*, and the second condition ensures *efficiency*.

[5] For a positive integer m, we use the notation $[m] \triangleq \{0, 1, \ldots, m-1\}$.

- Fast Generalized Array: Besides *load* and *store*, modern architectures like x86 commonly support read-modify-write (RMW) primitives, such as compare-and-swap (CAS), fetch-and-add (FAA), and fetch-and-store (FAS) [98]. In fact, some of these primitives are indispensible for efficiency and even solvability of problems that arise in concurrent systems. For instance, implementing a wait-free queue is impossible using only loads and stores [89]. Mutex locks can be implemented using loads and stores, but constant RMR (remote memory reference) complexity implementations are impossible using only loads and stores [17, 42, 152].

  Since RMW primitives are supported by hardware and are essential for concurrent algorithms, it would be ideal if the components of the fast array can be manipulated using these primitives. For instance, when implementing a fast array $\mathcal{O}$ on a multiprocessor that supports CAS and FAS in hardware, a process π should not only be able to read $\mathcal{O}[i]$ and write to $\mathcal{O}[i]$, but should also be able to apply CAS to $\mathcal{O}[i]$ and apply FAS to $\mathcal{O}[i]$. We term such an array, which allows all hardware-supported operations to be applied to its components, a *generalized array*.

  Let $\mathcal{S}$ be the set of hardware-supported RMW primitives. A *fast generalized array* is an implementation that not only supports $O(1)$-time linearizable Initialize(m, f), Read(i), and Write(i, v) operations, but also supports $O(1)$-time linearizable operations from the set $\mathcal{S}$.

In addition to defining the two types of concurrent fast arrays, our thesis makes the following four principal contributions:

- We design an algorithm for the (standard) fast array. If p processes share a fast array of length m, our algorithm uses only $O(m + p)$ space. More generally, to instantiate and use k fast arrays (for any k) of lengths $m_1, \ldots, m_k$, our algorithm uses only $O(M + p)$ space, where $M = \sum_{j=1}^{k} m_j$.

- We enhance the above algorithm to design a fast generalized array. Its space complexity is the same as the previous algorithm's—$O(m+p)$ for a single array of length m, and $O(M+p)$ for multiple arrays of combined length M.

- Based on the fast generalized array, we design a concurrent linearizable wait-free fixed-size hash table that supports expected constant time instantiation, insertion, and search.

- We benchmark the performance of the fast arrays.

Our algorithms require hardware support for read, write, and CAS; these instructions are available on most, if not all, modern hardware.

Efficiency lower bounds for concurrent data structures

My work on concurrent data structure lower bounds is present in Chapter 9.

We define a fundamental problem called *Generalized Wake-Up*—which captures the complexity of certain types of communication in shared memory systems that are fundamental to designing strongly consistent (linearizable) data structures—and show a lower bound on the number of shared memory instructions that need to be performed in order to solve this problem. Via reductions to *generalized wake-up*, we demonstrate that linearizable algorithms for fundamental objects like counters, queues, stacks, and priority queues must perform at least $\Omega(\log p)$ work per operation—even if the algorithms are randomized and even if the guarantees are amortized. For the union-find object, we show an $\Omega(\log \min\{n, p\})$ expected work lower bound on the cost of the single worst operation. Furthermore, we identify a class of "symmetric algorithms" that captures the complexities of all the known algorithms for the disjoint set union problem, and prove an $\Omega(m \cdot (\alpha(n, m/(np)) + \log(np/m + 1)))$ expected total work lower bound on algorithms of this class, thereby showing that our algorithm has optimal total work complexity for this class. Finally, we prove that any randomized algorithm, symmetric or not, cannot breach an $\Omega(m \cdot (\alpha(n, m/n) + \log\log(np/m + 1)))$ expected total work lower bound.

Original exposition on lower bounds in Telugu

Chapter 10 contributes, to the best of my knowledge, the first computer science research originally presented in the Telugu language. At the beginning of the chapter, I also present an abstract for the work in Sanskrit. In technical content, Chapter 10 is simply the Telugu original of Chapter 9.

Producing a scientific article in a new language required overcoming a unique set of challenges, including devising new terms for established and novel scientific concepts, and developing a XeLaTeX template to typeset mathematics and algorithms in Telugu. Scientific writing is fraught with technical terms, and even long established technical terms in English, had no equivalent in Telugu, a language in which computer science research papers have not previously been published. To overcome this barrier to expression, I had to develop terms in Telugu's mother language, Sanskrit—terms that were built from Sanskrit roots (dhātu), prefixes (upasarga), suffixes (pratyaya), and compounds (samāsa). Sanskrit terms can be immediately used in Indian Languages, including

Telugu, with standard, well-established alterations to their endings. Telugu is a technologically under resourced language, so I faced an additional roadblock occurred at the stage of typesetting the results. Out-of-the-box XeLaTeX—the variant of LaTeX that is used for typesetting non-Latin scripts—was not readily equipped to handle mathematical formulae and various environments in Telugu. Therefore, I developed a specialized *TeluguTeX* template in XeLaTeX that allowed for Telugu characters to appear in the mathematics environment, supported bold and emphasized text, and allowed for Telugu macros, citations, etc. I believe that these linguistic and technological solutions can have far reaching consequences in expressing other scientific work in Telugu and other languages. I present a greater discussion and specific future directions at the end of Chapter 14.

My article on multiprocessor algorithms in Telugu, which I reproduce with a Sanskrit abstract in Chapter 10, is to my knowledge, the first modern computer science research article in the language, which is spoken by over 80 million [189] and is the fastest growing language in America [21, 199]. I hope that this effort will spur more like it in Telugu and other languages, and that eventually, top quality scientific literature in native languages will affirm and enable many many more people to pursue their dreams of learning and contributing to the never ending corpus of science.

2.4 Contributions to Machine Verification

Data structures that organize, store, and quickly recall important pieces of information are the fundamental building-blocks behind fast algorithms. Thus, efficient and *rigorously proved* data structures are fundamental to reliable algorithm design. The task of designing such data structures for shared-memory multiprocessors, however, is notoriously difficult. Due to asynchrony, a t step algorithm for even just two processes has 2^t, i.e. exponentially many, possible executions depending on how the steps of the processes interleave. In fact, even deterministic concurrent algorithms have uncountably many possible infinite executions, as opposed to the single possible execution of a deterministic sequential algorithm. Designing algorithms that are correct in all of these executions is a grueling task, and thus, even mission critical concurrent code often suffers from subtle race conditions. For example: a subtle priority inversion bug in its concurrent code crashed the Pathfinder Rover days after its deployment on Mars and jeopardized the entire multi-million dollar NASA space mission [124]; a race in the power grid's energy management system stalled the alarm system for an hour, by which time it was too late to stop a cascading electrical outage that affected an estimated 55 million people in eight U.S. states and the province of Ontario, Canada [170]; and

the software of the radiation therapy machine, Therac-25, suffered from races that caused it to administer radiation doses that were over a hundred times as potent as the intended dose, which caused the deaths of at least three people and several more injuries [140, 142, 139]. Examples of errors in published concurrent data structures are also not left wanting [37, 53]. In this part, I develop techniques to obtain machine-verified correctness guarantees for concurrent algorithms; in particular, data structures.

In Part IV, we discuss our contributions to machine verification of multiprocess algorithms. Particularly, we focus on proofs of linearizability and strong linearizability.

Machine-Verifiable Proofs of Linearizability

My work on machine-verifiable proofs of linearizabilty and strong linearizability is presented in Chapter 11.

An object $\mathcal{O}$ is linearizable, if *for all* finite runs R of any algorithm $\mathcal{A}$ that uses $\mathcal{O}$, and every operation *op* that is performed on $\mathcal{O}$ in the run R, *there exists* a point in time between *op*'s invocation and return where it "appears to take place instantaneously". This definition, in particular the non-constructive existential quantifier "*there exists*" inside the universal quantification, makes it difficult to prove linearizability. This difficulty is only exacerbated, if the proof is to be provided in a way simple enough for a machine to verify. In fact, if approached naïvely, the prover would need to map each run of the algorithm to a linearization, i.e., a description of where in its invocation-response time interval each operation "appears to take place instantaneously", and then prove that each such mapping is legitimate. This is a difficult task, given that it is known that proving even a single fixed run R linearizable is NP-hard [74].

In this chapter, we devise a method for proving linearizability that not only works for a single implementation, or even a single type, but to devise a method that is *universal* and *complete*. By universal, we mean that our method should be powerful enough to allow for a proof of linearizability for implementations of *any* object type. By complete, we mean that *any linearizable implementation*, regardless of how complex its expression or linearization structure, must be provable by our method. Of course, our method will also be *sound*, meaning that any argument that is given using our method is indeed a correct mathematical proof of linearizability. Finally, we ensure that our method enables machine verifiable proofs by currently available proof assistants, which are generally built to verify proofs of simple program invariants.

1. We develop a rigorous *universal*, *sound*, and *complete* method for proving linearizability. In particular, we define a universal transformation that takes an arbitrary implementation $\mathcal{O}$, and outputs an algorithm $\mathcal{A}^*$, called the *tracker*, and a simple invariant $\mathcal{I}^*$, and prove a theorem that:

   $\mathcal{O}$ is a linearizable *if and only if* $\mathcal{I}^*$ is an invariant of $\mathcal{A}^*$.

   (Thus, we can produce a machine verified proof that $\mathcal{I}^*$ is an invariant of $\mathcal{A}^*$ to establish that $\mathcal{O}$ is linearizable.)

2. In fact, we give a whole family of transformations that each output different algorithms $\mathcal{A}'$, called *partial trackers*, with associated invariants $\mathcal{I}'$, and prove that for any of these partial trackers:

   $\mathcal{O}$ is a linearizable *if* $\mathcal{I}'$ is an invariant of $\mathcal{A}'$.

3. A close cousin of linearizability is *strong linearizability* [80], which ensures that even the hyper-properties of the data structure match those of an atomic object has also garnered a lot of recent interest [16]. We develop a rigorous *universal*, *sound*, and *complete* method for proving strong linearizability. In particular, we show that for each partial tracker $\mathcal{A}'$, there is an alternate associated invariant $\mathcal{I}''$, and we prove that:

   $\mathcal{O}$ is strongly linearizable *if and only if* some partial tracker $\mathcal{A}'$ has its associated $\mathcal{I}''$ as an invariant.

4. Finally, we demonstrate the power of our methods by producing machine-verified proofs of linearizability and strong linearizability for some notable data structures. In particular, we prove the linearizability and strong linearizability of our union-find object from Chapter 7, which is known to be the fastest algorithm for computing connected components on CPUs and GPUs [51, 95]. Our proof is verified by the proof assistant TLAPS (temporal logic of actions proof system) [153], and is publicly available on GitHub.[6] We and our collaborators have also used our method to produce TLAPS-verified publicly available linearizability proofs [211, 210, 94] of the Herlihy-Wing queue [93], and of Jayanti's single-writer single-scanner snapshot [105].

[6] proof available at: https://github.com/visveswara/machine-certified-linearizability

2.5 Contributions to Machine Learning

Machine learning is notorious for running very expensive computations on huge amounts of data, and thereby benefits extensively from fast and scalable parallel algorithms. We address the problem of speeding up Gibbs Sampling using multiprocessors in this thesis.

In Part V, we discuss our contributions to machine learning.

Asynchronous Gibbs Sampling

My work on asynchronous "HOGWILD!" Gibbs sampling is presented in Chapter 12.

Classical theories for data science generally model data sets as independent and identically distributed (iid) draws from a modelling distribution. However, many interesting real world data sets can be much too correlated to be treated under the iid assumption. Data provided by users from a social network is often correlated by the relations between the agents in the network, and data obtained from real-world sensors can be correlated depending on the proximity of the physical sensors and the time it was collected. In this work, we develop theoretical techniques to understand such types of non-independent data. In particular, we focus on high dimensional data and the question of distribution sampling.

The *distribution sampling* problem is a classical problem in computational statistics that sees applications in machine learning. The problem is to efficiently produce a sample $(X_1, \ldots, X_n)$ from a specified joint-distribution p. In the simplest case p is a distribution over $\{-1, +1\}^n$, with a probability mass function (pmf) of the form $p(x_1, \ldots, x_n) = \frac{1}{Z} \prod_{i,j} e^{\theta_{i,j} x_i x_j}$ for some constants $\theta_{i,j}$, where Z is chosen to make the sum of the masses equal to 1. (Such a distribution is called an *Ising Model*.) The problem sounds vexingly simple at first. The difficulty however arises since explicitly computing Z by summing takes exponential time.

The classical solution to the problem is sampling via Markov Chains. The idea is to run the *Gibbs Sampling* Markov Chain (Algorithm 1) of the distribution p to mixing, and use the resultant state vector $s = (s_1, \ldots, s_n)$ as an approximate sample. The efficiency of the process depends on the mixing time, T, of the Markov Chain. It is well known that the chain will be fast mixing (with $T = O(n \log n)$) if the distribution satisfies a certain condition known as *Dobrushin's Uniqueness*.

Algorithm 1 Gibbs Sampling

1: **Input:** Set of variables V, Configuration $x_0 \in S^{|V|}$, Distribution π initialization
2: **for** $t = 1$ to T **do**
3: Sample i uniformly from $\{1, 2, \ldots, n\}$;
4: Sample $X_i \sim \Pr_\pi [.|X_{-i} = x_{-i}]$ and set $x_{i,t} = X_i$;
5: For all $j \neq i$, set $x_{j,t} = x_{j,t-1}$;

While the $T = O(n \log n)$ mixing time is fast, there has recently been increased interest towards developing techniques for parallelizing this method to get even faster sampling times. Smola and Narayanamurthy proposed HOGWILD!-Gibbs, a lock-free asynchronous multiprocessor variant of Gibbs sampling [186]. The benefit of the HOGWILD! method is that each process needs to perform only T/p updates to the state vector in shared-memory. The algorithm however suffers from race conditions, and thus does not have the same strong guarantees as sequential Gibbs Sampling. More recently, De Sa et al. [48] proposed the study of HOGWILD!-Gibbs under a stochastic model of asynchrony in discrete graphical models. Most importantly, they show that the asynchronous Gibbs sampler accurately estimates events on any subset of the variables $\{X_i\}$ of size at most $O(\varepsilon n/\log(n))$, where ε is the total variational distance. Unfortunately, this result says nothing about statistics that depend on all the variables in a graphical model. Our results focus on this regime.

We push the understanding of HOGWILD!-Gibbs to new frontiers by proving that HOGWILD!-Gibbs samples can be used to estimate functions of *all the variables in a graphical model*. Under the same Dobrushin condition used in [48], and under a stochastic model of asynchrony with weaker assumptions, we show that one can do better than the bounds implied by [48] even for functions with bad Lipschitz constants. In particular, we show the following in our thesis:

- Starting at the same initial configuration, the executions of the sequential and the asynchronous Gibbs samplers can be coupled so that the expected Hamming distance between the multivariate samples that the two samplers maintain is bounded by $O(\tau \log n)$, where n is the number of variables in the graphical model, and τ is a measure of the average contention in the asynchrony model. More generally, the expectation of the d-th power of the Hamming distance is bounded by $C(d, \tau) \log^d n$, for some function $C(d, \tau)$.

- It follows that, if a function f of the variables of a graphical model is K-Lipschitz with respect to the d-th power of the Hamming distance, then the bias in the expectation of f introduced

by HOGWILD!-Gibbs is bounded by $K \cdot C(d, \tau) \log^d n$.

- Next, we improve the bounds for functions that are degree-d polynomials of the variables of the graphical model. Low-degree polynomials on graphical models are a natural class of functions which are of interest in many statistical tasks performed on graphical models (see, for instance, [46]). For simplicity we show these improvements for the Ising model, but our results are extendible to general graphical models. We show that the bias introduced by HOGWILD!-Gibbs in the expectation of a degree-d polynomial of the Ising model is bounded by $O((n \log n)^{(d-1)/2})$. This bound improves upon the Lipschitz bound by a factor of about $(n/\log n)^{(d-1)/2}$, as the Lipschitz constant with respect to the Hamming distance of a degree-d polynomial of the Ising model can be up to $O(n^{d-1})$. Importantly, the bias of $O((n \log n)^{(d-1)/2})$ that we show is introduced by the asynchrony is of a lower order of magnitude than the standard deviation of degree-d polynomials of the Ising model, which is $O((n)^{d/2})$ and is already experienced by the sequential sampler. Moreover, we also show that the asynchronous Gibbs sampler is not adding a higher order variance to its sample. Thus, our results suggest that running Gibbs sampling asynchronously leads to a valid bias-variance tradeoff.

  Our bounds for the expected Hamming distance between the sequential and the asynchronous Gibbs samplers follow from coupling arguments, while our improvements for polynomial functions of Ising models follow from a combination of our Hamming bounds and recent concentration of measure results for polynomial functions of the Ising model [45, 71, 81].

- We also illustrate our theoretical findings by performing experiments on a multicore machine. We experiment with graphical models over two kinds of graphs. The first is the $\sqrt{n} \times \sqrt{n}$ grid graph (which we represent as a torus for degree regularity) where each node has 4 neighbors, and the second is the clique over n nodes.

  First, we study how valid the assumptions of the asynchrony model are. The main assumption in the model was that the average contention parameter τ does not grow as the number of nodes in the graph grows. It is a constant which depends on the hardware being used and we observe that this is indeed the case in practice. The expected contention grows linearly with the number of processors on the machine but remains constant with respect to n. Next, we look at quadratic polynomials over graphical models associated with both the grid and clique graphs. We estimate their expected values under the sequential Gibbs sampler and

HOGWILD!-Gibbs and measure the bias (absolute difference) between the two. Our theory predicts that this should scale at $\sqrt{n}$ and we observe that this is indeed the case.

Chapter 3

Preliminaries

3.1 Model

In this thesis, we consider algorithms for *shared-memory multiprocess systems*. Such a system consists of a set Π of processes running programs concurrently on a machine with a shared random access memory. In addition to the shared memory, processes have private local registers. We model the processes as running *asynchronously*, meaning that we make no assumptions about the speeds at which individual processes take steps. More formally, asynchronicity is modeled as an *adversarial scheduler*, which decides which process gets to execute its next line of code at each time step of the concurrent system.

Synchronization Primitives Traditionally, registers in random access memory are known to support two atomic operations: load and store. Loading, a.k.a. reading, a shared memory register x returns the value of the register, and storing, a.k.a. writing, a value v to the register x changes its value to v. Modern multiprocessors offer several other atomic operations on shared memory registers. In this thesis, we use two of these operations: FAS (*fetch-and-store* a.k.a. *swap*) and CAS (*compare-and-swap*).

- Given a shared memory register x, a value v, and a private register r:

  $r \leftarrow \text{FAS}(x, v)$ instantaneously stores the current value of x to r and updates the value of x to v.

- Given a shared memory register x, two values u and v, and a private register r:

  $r \leftarrow \textsc{Cas}(x, u, v)$ instantaneously executes the following code. If the register x currently holds

the value u, then the CAS updates the value of x to v and sets r to *true*; otherwise, if x holds any other value, then the CAS does not affect the value of x and sets r to *false*.

Algorithm Execution In a *concurrent algorithm*, the system starts in some initial configuration C_0 and each process is given a program to execute, and the processes concurrently execute the steps of their individual programs as interleaved by the adversarial scheduler. In this context, some important terms of note are the following.

Definition 3.1.1 (step, event, run, history)**.**

- A *step* of an algorithm is a triple $(C, (\pi, \ell), C')$ such that C is a configuration, π is a process, ℓ is the line of code pointed to by π's program counter in C, and C' is a configuration that results when π executes line ℓ from C.
- The *event* corresponding to a step $(C, (\pi, \ell), C')$ is (π, ℓ), i.e., process π executing line ℓ.
- A *run* of an algorithm is a finite sequence $C_0, (\pi_1, \ell_1), C_1, (\pi_2, \ell_2), C_2, \ldots, (\pi_k, \ell_k), C_k$ or an infinite sequence $C_0, (\pi_1, \ell_1), C_1, (\pi_2, \ell_2), C_2, \ldots$ such that C_0 is an initial configuration and each triple $C_{i-1}, (\pi_i, \ell_i), C_i$ is a step.
- The *history* corresponding to a run is the subsequence of events in the run.

3.2 Data Objects

Many of the algorithms that I discuss in this thesis are for implementing concurrent data objects. Each data object is an implementation of some data type. Informally, the type specification describes the states a data type can be in, the set of operations the data type supports, and what state change and return value are prompted by each operation being performed in each state. Formally, a data type can be specified as follows.

Definition 3.2.1 (data type)**.** A *data type* τ consists of the following *components*:

- a set of *states* Σ that the object can be in.
- a set of *operations* OP that can be invoked on the object.
- for each $op \in OP$, a set of *arguments* ARG_{op} that the operation op can be called with.
- a set of *responses* RES, a.k.a. *return values*.

- a *transition function* $\delta(\sigma, \pi, op, arg)$ that outputs the new state σ' and the return value res that result when the operation op with argument arg is performed by process π while the object is in state σ. Formally, the transition function is

$$\delta : \Sigma \times \Pi \times \{(op, arg) \mid op \in OP, arg \in ARG_{op}\} \to \Sigma \times RES$$

Remark 3.2.2. Operations that require "no argument" (i.e., *read*), are modeled as taking an argument from a singleton set (i.e., $ARG_{read} = \{\perp\}$). Similarly, operations that return "no result" (i.e., *write*), are modeled as returning the result *ack*.

Remark 3.2.3. Our definition of δ (as a function) can be modified to a relation to allow various generalizations of the concept of object type—types that are non-deterministic, types where not every process is allowed to perform every operation (e.g., single writer snapshot), and types where an operation's behavior can depend on which process executes the operation (e.g., load-linked/store-conditional a.k.a., LL/SC).

Implementing complex objects, such as hash tables and union-find objects, from primitive objects supported by the underlying hardware (registers supporting read, write, CAS, etc.) is a central problem in multiprocessor programming. Below, we describe what an implementation entails. Later on, we will define what it means for an implementation to be correct, in the sense of *linearizability*.

Definition 3.2.4 (implementation)**.** An *implementation $\mathcal{O}$ of an object of type τ initialized to state σ_0* for a set of processes Π specifies

- A set of objects Ω called the *base objects* along with their types and initial states.
- A set of procedures $\mathcal{O}.op_\pi(arg)$ for each $\pi \in \Pi$, $op \in \tau.OP$, and $arg \in \tau.ARG_{op}$. The objects accessed in the code of the procedures must all be in Ω.

To execute an operation op with argument arg on the implemented object $\mathcal{O}$, a process π invokes the method $O.op_\pi(arg)$ (and executes the code in the procedure). The value returned by the method is deemed $\mathcal{O}$'s response to this operation invocation.

Implementation Behaviors Consider an object implementation $\mathcal{O}$, and a run R in which processes invoke operations on $\mathcal{O}$, execute the corresponding procedures of $\mathcal{O}$, and receive responses. By the definition of a run, R is an alternating sequence of configurations and events. Some of the

events are *invocation events*, i.e. calls to $\mathcal{O}$'s procedures, and some are *response events*, i.e. the execution of return statements of $\mathcal{O}$'s procedures. (Of course, there are other events, such as the execution of other lines between the call and return of a procedure.) We call the subsequence of R that includes only the invocation and response events the *behavior* in R. For example, if $\mathcal{O}$ is an initially empty queue, a behavior can be

$$(\pi_1, \textbf{invoke } enq_{\pi_1}(5)), (\pi_2, \textbf{invoke } deq_{\pi_2}()), (\pi_3, \textbf{invoke } enq_{\pi_3}(7)), (\pi_2, \textbf{response return } 7), (\pi_2, \textbf{invoke } enq_{\pi_2}(9))$$

Every possible behavior of an implementation $\mathcal{O}$ can be generated by the *generator algorithm* in the following figure, where each process repeatedly chooses an operation non-deterministically, invokes it by calling the corresponding procedure, and executes the procedure until it returns (receives a response). The next definition captures this discussion.

Initial Configurations:

- ω is an object of type τ, in its initial state σ_0.
- Each process $\pi \in \Pi$ is assigned the program $main_\pi()$; i.e. pc_π is initialized to the first line of main.
- Every other private register of each $\pi \in \Pi$ is initialized arbitrarily.

program $main_\pi()$

1: **while** *true* **do** choose any $(op, arg) \in \{(o, a) \mid o \in \tau.OP, a \in \tau.ARG_o\}$ and invoke $\mathcal{O}.op_\pi(arg)$

Figure 3-1: *Generator algorithm* $\mathcal{A}(\mathcal{O})$ that generates all behaviors of an implemented object $\mathcal{O}$ of type τ. Code shown for process $\pi \in \Pi$.

Definition 3.2.5 (implementation runs and behaviors)**.** Let $\mathcal{O}$ be an implementation of a type τ initialized to σ_0 for a set Π of processes. We define the *runs* of $\mathcal{O}$ to be the set of all runs of the *generator algorithm*, $\mathcal{A}(\mathcal{O})$. Let $\mathcal{R}$ be the set of all runs of $\mathcal{O}$. For any run $R \in \mathcal{R}$, we define $behavior(R)$ to be the subsequence of all the invocation and response events in R. The set of all *behaviors* of $\mathcal{O}$ is $\{behavior(R) \mid R \in \mathcal{R}\}$.

The Atomic Implementation Implementing an object from base objects of other types is often challenging, but implementing an object $\mathcal{O}$ from a base object ω of the same type is trivial: each procedure $\mathcal{O}.op_\pi(arg)$ is implemented simply by executing $\omega.op_\pi(arg)$ and returning the received response. We call this implementation the *atomic implementation*.

Definition 3.2.6 (atomic implementation)**.** The *atomic implementation* of an object $\mathcal{O}$ of type τ, initialized to σ_0, is the implementation presented in the following figure. (On line 2, the implementation resets r_π to $\perp$ as soon as it returns the value.)

Base Object: ω is an object of type τ, initialized to state σ_0.

procedure $\mathcal{O}.op_\pi(arg \in \tau.ARG_{op})$ ▷ for each $op \in \tau.OP$

1: $r_\pi \leftarrow \omega.op_\pi(arg)$

2: **return** r_π

$r_\pi \leftarrow \perp$

Figure 3-2: : Atomic implementation of $\mathcal{O}$ of an object of type τ initialized to state $\sigma_0 \in \tau.\Sigma$.

Linearizability We are now ready to define *linearizability*. Intuitively, an object implementation is linearizable if it behaves like an atomic object of the same type. Formally:

Definition 3.2.7 (linearizability)**.** For a set Π of processes, let $\mathcal{O}$ be an implementation of an object of type τ initialized to σ_0, and let $\mathcal{O}_{atomic}$ be the atomic implementation of an object of type τ initialized to σ_0. Furthermore, let $\mathcal{R}$ be the set of all runs of $\mathcal{A}(\mathcal{O})$ and $\mathcal{R}_{atomic}$ be the set of all runs of $\mathcal{A}(\mathcal{O}_{atomic})$. We say a run $R_{atomic} \in \mathcal{R}_{atomic}$ is a *linearization* of a run $R \in \mathcal{R}$ if $behavior(R) = behavior(R_{atomic})$. Correspondingly, we say that a run $R \in \mathcal{R}$ is *linearizable* if it has a linearization $R_{atomic} \in \mathcal{R}$; equivalently, if $behavior(R)$ is also a behavior of O_{atomic}. We say that the implementation $\mathcal{O}$ is linearizable if every finite run $R \in \mathcal{R}$ is linearizable. Equivalently, $\mathcal{O}$ is linearizable if every finite behavior of $\mathcal{O}$ is a behavior of $\mathcal{O}_{atomic}$.

Strong Linearizability In general, a run R of a linearizable implementation can have multiple linearizations. Intuitively, a linearizable object implementation satisfies *strong linearizability* if for any run of the implementation R the object can "commit to a specific linearization" $\mathcal{L}(R)$, such that the linearization of any extension of the run R is an extension of $\mathcal{L}(R)$.

Definition 3.2.8. For a set Π of processes, let $\mathcal{O}$ be an implementation of an object of type τ initialized to σ_0, and let $\mathcal{O}_{atomic}$ be the atomic implementation of an object of type τ initialized to σ_0. Furthermore, let $\mathcal{R}$ be the set of all runs of $\mathcal{A}(\mathcal{O})$ and $\mathcal{R}_{atomic}$ be the set of all runs of $\mathcal{A}(\mathcal{O}_{atomic})$. An implementation is *strongly linearizable*, if there is a *linearization function* $\mathcal{L} : \mathcal{R} \to \mathcal{R}_{atomic}$ that

maps each run R of the implementation to an atomic run $\mathcal{L}(R)$ of the atomic implementation that is a linearization of it, such that if R_{pre} is a prefix of R, then $\mathcal{L}(R_{pre})$ is a prefix of $\mathcal{L}(R)$.

Not all linearizable implementations are strongly linearizable, since, for some implementations, we need to extend a different linearizations of a run R to linearize different extensions of R. This notion of *strong* linearizability is subtle, but has been shown important in preserving hyperproperties of composed algorithms, such as output probability distributions [80, 16].

3.3 Complexity Measures

Work Complexity The most prevalent efficiency measure for sequential algorithms is time complexity. The classic analog of this efficiency measure in multiprocess computing is *work complexity*. Specifically, we say that each atomic instruction executed by a process costs one unit of *work* . The *work done by a process* π is the number of units of work performed by π, and the *total work* done by all processes is the sum of the work done by a process over all processes. Work complexity in multiprocessors is so analogous to time complexity in uniprocessors, that colloquially people refer to "work" as "time" and "work complexity" as "time complexity".

RMR (Remote Memory Reference) Complexity While work complexity accounts for all instructions performed, sometime we wish to only account for the particularly expensive instructions that are performed on *remote memory*. Whether a given instruction is a *Remote Memory Reference (RMR)* or not, depends on the machine model. There are two machine models: *Cache Coherent* (CC) and *Distributed Shared Memory* (DSM).

In the CC model, all shared variables reside in memory, which is considered remote to all processes. Additionally, each process has a local cache where copies of shared variables can reside. When a process π reads a shared variable x, x's copy is brought into π's cache if it is not already there. When π performs a non-read operation on x, copies of x in all caches are deleted. Thus, a read operation by π on a shared variable that is in π's cache has no need to access the network, but every other read operation and every non-read operation accesses the network and is counted as a *remote memory reference* (RMR).

In the DSM model (aka NUMA) there are no caches, but the shared memory is partitioned so that each process is assigned one part that resides locally at that process. A shared variable x resides permanently in some process' partition, but every process π can perform operations on x,

regardless of whether x resides in π's partition or not. A (read or a non-read) operation by π on x is counted as an RMR if and only if x does not reside in π's partition of shared memory.

Part II

Mutual Exclusion Locks

Chapter 4

Standard Mutual Exclusion

4.1 Introduction

Mutual Exclusion calls for the design of an algorithm by which multiple asynchronous processes can compete with each other to acquire a lock and release it. It is the oldest and a fundamental problem in distributed computing [52], and is modeled as follows. Each process repeatedly cycles through four sections of code—Remainder, Try, Critical, and Exit. A process stays in the Remainder when it does not need the lock and, when it wants it, it executes the Try section, concurrently with others that are also competing for the lock. When the Try section terminates, the process enters the Critical Section (CS), where it has the exclusive ownership of the lock. When the process no longer needs the lock, it gives the lock up by executing the Exit section and moves back to Remainder.

The *mutual exclusion problem* consists of designing the code for the Try and Exit sections so that the first property and one of the two versions of the next property in the following list are satisfied. The last two in the list are desirable, but not always required.

- Mutual Exclusion: At most one process is in the CS at any time.
- Live-Lock Freedom (resp. Starvation-Freedom): Suppose that each process that enters the CS eventually leaves the CS, and no process stops taking steps while in the Try or Exit sections. Then, (i) if a process p is in the Try section, *some process* (resp. *process* p) will be in the CS at a later time, and (ii) if a process p is in the Exit section, *some process* (resp. *process* p) will be in the Remainder at a later time.

- Bounded Exit: There is a bound b such that each process in the Exit section completes that section in at most b of its own steps.

- First-Come-First-Served (FCFS): The Try section code is split into a bounded doorway, followed by a waiting room. If a process p completes the doorway before a process q enters the doorway, then q does not enter the CS before p.

For the first two decades following Dijkstra's original paper on this topic, algorithms were designed for the time sharing *uni*processors of that era [174]. It was later recognized that, to perform well on *multi*processors, the traffic that an algorithm generates on the interconnection network that connects processors and memory modules should be minimized. This observation spurred research on designing efficient algorithms for multiprocessors, which are commonly modeled as either *Cache Coherent* (CC) machines or *Distributed Shared Memory* (DSM) machines.

In the CC model, all shared variables reside in memory, which is considered remote to all processes. Additionally, each process has a local cache where copies of shared variables can reside. When a process p reads a shared variable x, x's copy is brought into p's cache if it is not already there. When p performs a non-read operation on x, copies of x in all caches are deleted. Thus, a read operation by p on a shared variable that is in p's cache has no need to access the network, but every other read operation and every non-read operation accesses the network and is counted as a *remote memory reference* (RMR).

In the DSM model (aka NUMA) there are no caches, but the shared memory is partitioned so that each process is assigned one partition that resides locally at that process. A shared variable x resides permanently in some process' partition, but every process p can perform operations on x, regardless of whether x resides in p's partition or not. A (read or a non-read) operation by p on x is counted as an RMR if and only if x does not reside in p's partition of shared memory.

With either model, the *worst-case RMR complexity* of an algorithm is the maximum number of RMRs incurred by a process between the times of leaving and subsequently re-entering the Remainder section. The *amortized RMR complexity* of an algorithm is the maximum value of x/y, where x is the total number of RMRs performed by all processes and y is the total number of times the Try section was entered by any process. In general, the RMR complexity is function of n, which is the maximum number of processes that execute the algorithm. Unless otherwise specified, we use "RMR complexity" to refer to "worst-case RMR complexity".

It is well known that the RMR complexity is crucially dependent on the set of hardware atomic

instructions available to manipulate the shared memory. Modern multiprocessors support instructions, besides *read* and *write*, such as *Compare-And-Swap* (CAS) and *Fetch-And-Store* (FAS). The instruction CAS(X, u, v) compares memory word X's value with u. If X's value is u, it changes X's value to v and returns *true*; otherwise it returns *false*, without changing X's value. The instruction FAS(X, v) applied to the memory word X that contains some value u, stores v in X and returns u.

It turns out that CAS is not a suitable primitive to design constant RMR locks: if only read, write, and CAS may be used, Cypher proved that constant RMR complexity is not achievable [42], and more recently, Attiya et al. proved that not even sub-logarithmic RMR complexity is achievable [17]. Thus, it is imperative that any algorithm that aspires to achieve $O(1)$ RMR complexity employs some instruction different from CAS.

4.1.1 The MCS lock and its advantages

An ideal mutual exclusion algorithm would have $O(1)$ RMR complexity, i.e., the number of RMRs performed by a process in Try and Exit sections is a constant, *independent* of how many processes contend for the lock. Anderson's algorithm was the first to meet this ideal, but it has $O(1)$ RMR complexity only for the CC model [13]. The watershed moment came subsequently when Mellor-Crummey and Scott designed an algorithm [152], popularly known as the *MCS lock*, that not only achieves $O(1)$ RMR complexity for both CC and DSM models, but also has the following wonderful properties:

- Support for an arbitrary set of processes: The MCS algorithm can be executed concurrently by an arbitrary number of processes with arbitrary names, in contrast to some algorithms that support only N processes, where N is fixed at the time of designing the algorithm.

- Space frugality: When N processes share an MCS lock, the total memory requirement is only $O(N)$ words. In fact, the MCS algorithm satisfies the following stronger property that we call *space frugality*.

  The MCS algorithm can be used to implement L locks, shared by N processes, using only $O(L + N)$ words, provided that no process holds on to more than one lock at a time (i.e., each process relinquishes its ownership of a lock before acquiring the ownership of another lock). It is noteworthy that only $L + N$ words are needed, and not LN words.

- Statelessness: The MCS algorithm is *stateless*, i.e., once a process executes Try, CS, and Exit

and returns to Remainder, there is nothing about the lock that the process needs to remember while in the Remainder section.

Table 4.1: Comparison of Existing Mutual Exclusion Algorithms

	$O(1)$ **RMR for DSM**	**Bounded Exit**	**FCFS**	**Stateless**	**Space Frugal**	**Supports Arbitrary Set of Processes**	**Single Special Instruction**	**Other Drawbacks**
Anderson [13]	No				No	No		
Graunke, Thakkar [83]	No							
MCS [152]		No					FAS & CAS	
Craig [41]								not memory safe
Magnusson et. al [147]	No							
Rhee [176]		No		No	No			
Lee Alg.1 [138]				No	No	No		uses $\Theta(N^2)$ space
Lee Alg.2 [138]		No		No	No	No		
Anderson, Kim [10]			No		No	No		
Dvir et. al Alg.1 [56]				No	No		FAS & CAS	
Dvir et. al Alg.2 [56]							FAS & CAS	
This chap (main)								
This chap (toggling)				No	No			
This chap (switching)								

4.1.2 A shortcoming of the MCS lock

The MCS algorithm does not satisfy the important Bounded Exit property: a process in the Try section can cause an exiting process to wait, which is undesirable because there is no intrinsic reason why a process should be obstructed by others from giving up the lock.

4.1.3 Overcoming shortcomings: node-switching and node-toggling strategies

The MCS algorithm maintains a queue of nodes, one for each waiting process. When a process p leaves the CS, it checks if its node x is the last in the queue. If the answer is no because some other process q enqueued its node y behind x, p waits until q communicates to p its "spin-location" (the location where q will wait for p's permission to enter the CS). This waiting by p is the reason that the MCS Exit section is not bounded. Craig eliminated this waiting by requiring each exiting process to deposit in its node a "token" that will be picked up by the next process in the queue [41]. Thus, as p leaves the CS, instead of waiting for q, p deposits a token in x and lets x remain in the queue. When depositing the token, if p sees that q has already left behind its spin location in x, p will simply go to that location and inform q that it can enter the CS. This coordination, since it involves only two processes p and q, is possible using only FAS (there is no need for CAS). All is well with this strategy except that p, which owned x when it entered the protocol, had to leave x behind in the queue when exiting the protocol. To prevent each process from having to use a

new node each time it executed the protocol, Craig made a clever observation: once q, the process that enqueues y behind x consumes the token in x, x is no longer needed in the queue. So, when q exits the protocol, even as it leaves y behind in the queue (having deposited a token in y), it grabs the node x and uses x the next time it executes the protocol. We call this the "node-switching" strategy because q came in with node y and switched to x as it left the protocol.

An alternative is the "node-toggling" strategy introduced by Rhee [176]. Here too the idea is for p to deposit a token in its node x and leave x behind in the queue when returning to the Remainder section. However, unlike in the previous strategy, p won't grab x's predecessor. Instead, each process owns two nodes (the same two nodes at all times) and, when p executes the protocol the next time, it simply uses the other node that it owns—the one different from x. We call this the "node-toggling" strategy as each process toggles between its two nodes each time it executes the protocol.

4.1.4 Our contribution

Queue lock is a generic term that refers to any mutual exclusion algorithm, such as MCS, that organizes the waiting processes in a FIFO queue. Since the time MCS lock was introduced, many queue locks were proposed to overcome its shortcoming, but they either lacked some strengths of MCS or introduced a new drawback, as we explain in the next section.

In this chapter, we present a queue lock that we believe is the first to have all of the advantages of the MCS lock and eliminate its shortcoming, without introducing any new undesirable features. In particular, our algorithm has a bounded Exit section, while still ensuring $O(1)$ RMR complexity on both CC and DSM models, support for an arbitrary number of processes with arbitrary names, space-frugality and statelessness. Moreover, unlike MCS, which requires hardware support for both CAS and FAS instructions, our algorithm requires support for only FAS.

A further highlight is that, through two simple instantiations of our algorithm, we derive both a node-switching algorithm and a node-toggling algorithm that are the simplest in their respective classes among the known algorithms.

Our algorithm is also accompanied by a rigorous, invariant-based proof of correctness.

4.1.5 Comparison to prior research

The MCS lock is an example of a "queue lock," which is a generic term for any mutual exclusion algorithm in which waiting processes organize themselves in a queue. In the table below, we compare

all the queue locks of $O(1)$ RMR complexity that we are aware of, starting from the earliest one by Anderson [13] to the recent one by Dvir and Taubenfeld [56], against the criteria discussed above. We note that array based algorithms [13] and node-toggling algorithms [176, 138, 56] are inherently not space adaptive. Some of the node-switching algorithms, as described in their papers, are not space adaptive, but since they can be easily modified to be space adaptive, we have counted them as satisfying this property.

Craig designed the earliest algorithm that satisfies all of the good properties we have listed [41], but as pointed out by Dvir and Taubenfeld [56] and as we explain now, his algorithm has one undesirable feature. When each node in the queue points to the next node, Craig's algorithm needs to pack a control bit together with the "next" pointer into the same memory word. Consequently, the full width of memory words is not made available to store pointers.

The second algorithm in Dvir and Taubenfeld's work also satisfies all of the good properties we have listed [56]. However, unlike our algorithm, which requires support for only FAS, their algorithm relies on both FAS and CAS. More importantly, their use of four fields per node, compared to just a single field in our algorithm, led to a more complex, 15-line algorithm. In comparison, our node-switching algorithm is only 8 lines long.

4.2 Main Algorithm

4.2.1 Informal Description

The lock described in our algorithm is a *queue lock*, i.e. all k processes that have passed through the doorway will queue in the order in which they finished the doorway $p_1, \ldots, p_k$. p_1 will be called the *head* process and p_k the *tail* process. For a given process p_i in the queue, p_{i-1} is its *predecessor* and p_{i+1} its *successor*. If such processes do not exist we will say that p_i does not have a predecessor/successor in the queue.

We will implement this queue ordering by giving each process p two associated variables: $mynode_p$, which points to process p's node in shared memory, and $prev_p$. For the ordering above, for all $i \in [2, k]$, $prev_{p_i} = mynode_{p_{i-1}}$ and $prev_{p_1}$ points to the *anchornode*, a special node that is no active process's *mynode*. The shared variable TAIL maintains a pointer to the tail node, $mynode_{p_k}$, to allow new entering processes to find the end of the queue to extend it. In general, the head process p_1 is in the CS (or enabled to enter the CS). When it wishes to exit, it frees the current *anchornode*, enables its successor, and leaves for the Remainder Section. So, the old $mynode_{p_1}$

Algorithm 2 Our Main Algorithm

Shared Variables

sentinel: a pointer, initialized to any non-nil value

TAIL: holds a pointer, initially TAIL = *sentinel*

When a process p joins the protocol it allocates memory for:

NODE_p: holds a pointer, arbitrarily initialized

GO_p: a boolean in p's partition of shared memory, arbitrarily initialized

Local Variables

For each process p:

$mynode_p$: holds a pointer (to a NODE), initially $mynode_p = \&\text{NODE}_p$

$prev_p$: holds a pointer (to a NODE), arbitrarily initialized

go_p: holds a pointer to GO_p

$next\_go_p$: holds a pointer (to a *go* variable), arbitrarily initialized

1: $mynode_p \leftarrow$ **malloc**(shared word) ; $go_p \leftarrow$ **malloc**(shared boolean in p's partition)
2: $*mynode_p \leftarrow$ `nil`
3: $*go_p \leftarrow$ *false*
4: $prev_p \leftarrow \text{FAS}(\text{TAIL}, mynode_p)$ ▷ Doorway Ends
5: **if** $\text{FAS}(*prev_p, go_p) =$ `nil` **then**
6: **wait till** $*go_p =$ *true*
CRITICAL SECTION
7: **free**($prev_p$); **free** (go_p)
8: **if** $(next\_go_p \leftarrow \text{FAS}(*mynode_p, \text{NONNIL})) \neq$ `nil` **then** ▷ NONNIL can be any value not equal to `nil`
9: $*next\_go_p \leftarrow$ *true*

becomes the new *anchornode*, and for each $i \in [2, k]$, p_i becomes p_{i-1}.

A new process p that is interested in entering the CS enters the protocol by allocating itself a *node*—a single word in shared memory—that is pointed to by its local variable $mynode_p$, and a *go variable*—a single boolean in shared memory—that is pointed to by its local variable go_p (Line 1). It initializes $*mynode_p$ to `nil` and $*go_p$ to *false* (Line 2+3), and then enters the queue by swapping $mynode_p$, into TAIL and assigning $prev_p$ the result, the previous last node in the queue (Line 4). If the queue was previously empty, making p the head process, the swap would return the *anchornode* which would hold a non-nil value, otherwise, $prev_p$ would point to p's predecessor's *mynode* which would hold `nil`. The simplest algorithm therefore would make p busy-wait until $*prev_p \neq$ `nil`, and an exiting process q would simply write a non-nil value to $*mynode_q$ to enable its successor. However, $prev_p$ points to a node of another process and thus busy-waiting on its value would lead to unbounded RMR on a DSM machine. So, instead of reading $*prev_p$ multiple times, p FASs $*prev_p$ with go_p (Line 5). We call this the entry-swap. If the entry-swap returns a non-nil value, p

knows it is the head and can thus proceed into the CS, otherwise it busy-waits on $*go_p$ which is in its own partition of shared memory (Line 6). Reciprocally, an exiting process q will free the current anchor node (Line 7) and will then perform an exit-swap—it will FAS $*mynode_q$ with a non-nil value. If the exit-swap returns `nil`, q knows that its successor has not attempted its entry-swap yet and is thus enabled. Otherwise, the return value of the exit-swap will be the *go* boolean on which q's successor is busy-waiting; so, q simply sets that boolean to true before going to the Remainder Section (Line 8-9).

4.2.2 Invariant

In this section we will prove the algorithmic invariant—a statement about the algorithm that holds true in any configuration of the multiprocess system. The invariant will be a key building block in proving the various properties of the algorithm, i.e., *mutual exclusion, starvation freedom, FCFS, etc.*

First, we define the following sets that will need to be referred to repeatedly.

$$P = \{p \mid PC_p \in [2, 9]\} \qquad \text{(Active Processes)}$$

$$R = \{p \mid PC_p \in [5, 9]\} \qquad \text{(Registered Processes)}$$

$$\mathcal{N} = \{mynode_p \mid p \in P\} \qquad \text{(Nodes of Active Processes)}$$

$$\mathcal{A} = \mathcal{N} \cup \{prev_p \mid PC_p \in [5, 7]\} \cup \{\text{TAIL}\} \qquad \text{(Allocated Nodes)}$$

P is the set of all *active processes*, while R is the subset of these processes that have already *registered* themselves in the wait-queue by performing the first FAS instruction on the TAIL. $\mathbb{N}$ is the set of nodes owned by the active processes. We have also defined a set of nodes $\mathcal{A}$. Our invariant will prove that $\mathcal{A}$ is the set of *allocated nodes*, i.e., the set of nodes that have been allocated by a **malloc**, but that have not yet been deallocated by a **free**. (We consider *sentinel* to be initially allocated.)

We now present the rigorous mathematical statement of the invariant in Figure 4-1. To aid readability, we also provide below an informal description of the invariant.

1. Every active process $p \in P$ has a unique non-nil variable $mynode_p$.

2. If there are no processes in the wait-queue (i.e. $|R| = 0$), then TAIL holds a special node that is no active process's node. And, $*$TAIL holds a non-nil value. (So, the first process to register

itself can go to the Critical Section.)

3. If there are $k > 0$ registered processes in the wait-queue they can be ordered $p_1, \ldots, p_k$ from first to last such that

   (a) If p_1 is still in the waiting room, then its predecessor node $prev_{p_1}$ is non-nil and is not any process q's $mynode_q$.

   (b) p_1 is either in the Critical Section, Exit Section, or enabled to enter the Critical Section and in the waiting room.

   (c) If p_1 is on line 9, then there is at least one more process in the wait-queue, p_1 has its successor p_2's *go* variable in its $next\_go_{p_1}$ field, and p_2 is on line 6, waiting for go_{p_2} to become true.

   (d) If p_2 is waiting on line 6 for its go variable to become true, then either p_1 is already on line 9, or it will get to line 9 and obtain p_2's *go* address, since $*mynode_{p_1} = go_{p_2}$.

   (e) For $i \in [2, k]$

      i. The processes in the wait-queue form a chain with $prev_{p_i} = mynode_{p_{i-1}}$.

      ii. Process p_i is not enabled since $*go_{p_i} = false$.

      iii. Process p_i is in the waiting room (on line 5 or 6).

      iv. If p_i is on line 5, it is not enabled since $*mynode_{p_{i-1}} = \texttt{nil}$.

      v. If p_i is on line 6, then it has swapped its go address into its predecessor's node contents.

   (f) TAIL points to $mynode_{p_k}$ the last node in the wait-queue of nodes.

   (g) *TAIL holds `nil`.

4. The *mynode* and *go* variables are correctly initialized to `nil` and *false* by lines 3 and 4.

5. $\mathcal{A}$ is the set of nodes that have been allocated by **malloc** and not yet deallocated by **free**. (We consider *sentinel* to be initially allocated.)

Lemma 4.2.1. *The invariant I presented in Figure 4-1 holds at each step of the algorithm.*

Proof of Lemma 4.2.1. The proof is by induction on the steps taken by the multiprocess system.

1. $\forall p, q \in P, ((p \neq q \implies mynode_p \neq mynode_q) \wedge (mynode_p \neq \texttt{nil}))$
2. If $R = \varnothing$, then $(*\text{TAIL} \neq \texttt{nil} \wedge \text{TAIL} \notin \mathcal{N} \cup \{\texttt{nil}\})$
3. If $k = |R| > 0$, there is an order $p_1, p_2, \ldots, p_k$ of the processes in R such that
   (a) $PC_{p_1} \in [5, 7] \implies prev_{p_1} \notin \mathcal{N} \cup \{\texttt{nil}\}$
   (b) $(PC_{p_1} \in [7, 9]) \vee (PC_{p_1} = 6 \wedge *go_{p_1} = true) \vee (PC_{p_1} = 5 \wedge *prev_{p_1} \neq nil)$
   (c) $PC_{p_1} = 9 \implies (k \geq 2 \wedge PC_{p_2} = 6 \wedge next\_go_{p_1} = go_{p_2})$
   (d) $(k \geq 2 \wedge PC_{p_2} = 6) \implies ((PC_{p_1} \neq 9 \wedge *mynode_{p_1} = go_{p_2}) \vee (PC_{p_1} = 9))$
   (e) $\forall i \in [2, k]$:
      i. $prev_{p_i} = mynode_{p_{i-1}}$
      ii. $*go_{p_i} = false$
      iii. $PC_{p_i} \in [5, 6]$
      iv. $PC_{p_i} = 5 \implies (*mynode_{p_{i-1}} = \texttt{nil})$
      v. $[(i > 2 \wedge PC_{p_i} = 6) \implies (*mynode_{p_{i-1}} = go_{p_i})] \wedge$
      $[(PC_{p_1} \neq 9 \wedge PC_{p_2} = 6) \implies (*mynode_{p_1} = go_{p_2})]$
   (f) $\text{TAIL} = mynode_{p_k}$
   (g) $*\text{TAIL} = \texttt{nil}$
4. $\forall p, (PC_p = 3 \implies *mynode_p = \texttt{nil}) \wedge (PC_p = 4 \implies (*go_p = false \wedge *mynode_p = \texttt{nil}))$
5. $\mathcal{A}$ is the set of nodes that have been allocated but not yet freed.

Figure 4-1: Invariant I is the main invariant of Algorithm 2.

Base Case: In the initial state, all processes are in the Remainder Section, so $P = R = \mathcal{N} = \varnothing$. Since TAIL = *sentinel* initially, $\mathcal{A} = \{\text{TAIL}\} = \{sentinel\}$. Each part of the invariant I holds for the following reasons.

1. I_1 holds trivially since $P = \varnothing$.
2. I_2 holds since $\text{TAIL} = sentinel \notin \{\texttt{nil}\} = \mathcal{N} \cup \{\texttt{nil}\}$ and since $*\text{TAIL}$ is initialized to NONNIL.
3. I_3 holds trivially since $|R| \not> 0$.
4. I_4 holds trivially since $PC_p \notin \{3, 4\}$ for every process p.
5. I_5 holds since $\mathcal{A} = \{\text{TAIL}\} = \{sentinel\}$ initially and the *sentinel* is the only node that has been allocated and not yet freed.

Induction Step: We now assume that the invariant holds in a given configuration of the multiprocess system, and prove that it will hold after exactly one more step is executed by

some process. Namely, we consider what happens when a process π executes line l. We use unprimed names to denote sets before π's execution of l and primed variables to denote the same sets after the line execution.

line 1: We notice that $\pi \notin P$ before the line execution, and by definition $P' = P \cup \{\pi\}$, $\mathbb{N}' = \mathcal{N} \cup \{mynode_\pi\}$, $\mathcal{A}' = \mathcal{A} \cup \{mynode_\pi\}$, and $R = R'$.

- I_2, I_3, and I_4 continue to hold since $R' = R$ and the set of processes p with $PC_p \in [3, 4]$ are unaffected by the line execution.
- Let $p, q \in P'$, and $p \neq q$. If $p, q \in P$, then $mynode_p \neq mynode_q$ since I_1 held before the execution of l. If $p = \pi$, then $mynode_p \neq mynode_q$ since **malloc** returns a shared word that is currently unallocated, and $mynode_q$ is in the set of allocated nodes $\mathcal{A}$ before the execution of l. Furthermore, $mynode_p \neq$ `nil` since I_1 held before the execution of l and $mynode_\pi$ is guaranteed to be non-nil by **malloc**. Therefore, I_1 continues to hold.
- $\mathcal{A}' = \mathcal{A} \cup \{mynode_\pi\}$ and π's line execution newly allocated $mynode_\pi$ and freed nothing.

line 2:
- I_1, I_2, I_3, and I_5 continue to hold as they are unaffected by this line.
- I_4 continues to hold since (i) every process $p \in [3, 4]$ and $p \neq \pi$ already satisfied the invariant and is unaffected by π's execution of line 2, and (ii) $PC_\pi = 3$, and $*mynode_\pi =$ `nil` by the execution of line 2.

line 3:
- I_1, I_2, I_3, and I_5 continue to hold as they are unaffected by this line.
- I_4 continues to hold since (i) every process $p \in [3, 4]$ and $p \neq \pi$ already satisfied the invariant and is unaffected by π's execution of line 3, and (ii) $PC_\pi = 4$, and $*mynode_\pi =$ `nil` by the I_4 before the execution of line 3, and $*go_\pi = false$ by π's execution of line 3.

line 4: $\pi \notin R$, and $R' = R \cup \{\pi\}$ by definiton.

- I_1, I_4, and I_5 continue to hold as they are unaffected by this line.
- I_2 continues to hold as it is now trivial (since $|R'| \geq 1$)
- We now turn our attention to I_3 the only remaining part of the invariant to be proved. Let $k = |R'|$, and note that $|R| = k - 1$. Note that $\pi = p_k$ in the process ordering. I_{3f} holds in all cases since the FAS operation in the executed line 4 made

TAIL $= mynode_\pi$ and $p_k = \pi$. I_{3g} holds since $*mynode_\pi$ was `nil`before the line execution by I_4, and TAIL $= mynode_\pi$ by I_{3f}.

We now consider three cases, $k = 1, k = 2$, and $k > 2$ to show the rest of the parts of I_3.

$k = 1$ $PC_{p_1} = PC_\pi = 5$.

I_{3c} holds since $PC_{p_1} \notin [7, 9]$. I_{3d}, and I_{3e} hold trivially since $k < 2$.

I_{3a} holds since $prev_{p_1}$ is the old value of TAIL, which was not in $\mathcal{N} \cup \{\texttt{nil}\}$ by I_2 (since $|R| = 0$).

I_{3b} holds since $PC_{p_1} = 5$ and $*prev_{p_1} \neq \texttt{nil}$ since the old value of $*$TAIL was non-nil by I_2.

$k = 2$ I_{3a}, and I_{3b} continue to hold since they were unaffected by the execution of the line.

I_{3c} continues to hold since PC_{p_1} could not have been 9 before the line execution—since $|R|$ was 1—and the value of p_1's program counter did not change.

I_{3d} holds since $PC_{p_2} = PC_\pi = 5 \neq 6$.

We remind ourselves that $\pi = p_2$. The five subparts of I_{3e} hold since:

i. TAIL was $mynode_{p_1}$ by I_{3f} and that value was swapped into $prev_{p_2}$.

ii. $*go_{p_2} = *go_\pi = \mathit{false}$ since PC_π was 4 before the execution of the line, and therefore I_4 imposed this condition.

iii. $PC_{p_2} = 5 \in [5, 6]$.

iv. Since $PC_{p_2} = 5$, I_{3c} shows that $PC_{p_1} \neq 9$, and therefore I_{3d} imposes that $*mynode_{p_1} = \texttt{nil}$. Since $prev_{p_2} = mynode_{p_1}$ by I_{3ei}, the proof is complete.

v. Holds trivially since $PC_{p_2} = 5 \neq 6$.

$k > 2$ $I_{3a}, I_{3b}, I_{3c}, I_{3d}$, and the five subparts of I_{3e} for $i < k$ are unaffected by the line execution.

The five subparts of I_{3e} hold when $i = k$ since:

i. TAIL was $mynode_{p_{k-1}}$ by I_{3k} and that value was swapped into $prev_{p_k}$ by the execution of line 4 by $\pi = p_k$.

ii. $*go_{p_k} = *go_\pi = \mathit{false}$ since PC_π was 4 before the execution of the line, and therefore I_4 imposed this condition.

iii. $PC_{p_k} = 5 \in [5, 6]$.

iv. $*mynode_{p_{k-1}} = \texttt{nil}$ by I_{3g}, since TAIL was equal to $\text{NODE}_{p_{k-1}}$ before the line execution.

v. Holds trivially since $PC_{p_k} = 5 \neq 6$.

line 5:
- I_1, I_2, I_4, and I_5 continue to hold as they are unaffected by this line.
- To prove I_3, we consider the cases $\pi = p_1$, $\pi = p_2$, and $\pi = p_i$, where $i > 2$.

$\pi = p_1$ By I_{3b} before the execution of the line, $*prev_{p_1} \neq \texttt{nil}$ so p_1 will enter the CS and be at line 7 after the execution.

I_{3a} holds since $prev_{p_1}$ is unaffected by the line.

I_{3b}, and I_{3c} hold since $PC_{p_1} = 7 \neq 9$ after the line.

I_{3d} holds since $k \geq 2 \wedge PC_{p_2} = 6$ implies that $*mynode_{p_1} = go_{p_2}$ by I_{3ev}.

I_{3e}, I_{3f}, and I_{3g} continue to hold as they are unaffected by this line execution.

$\pi = p_2$ By I_{3eiv} and I_{3ei} the comparison will succeed since $*prev_{p_2} = *mynode_{p_1} = \texttt{nil}$ and go_{p_2} will be successfully placed into $*mynode_{p_1}$. Furthermore, PC_{p_2} will become 6.

$I_{3a}, I_{3b}, I_{3c}, I_3f, I_{3g}$, and all subparts apart from (v) are unaffected by the line.

I_{3ev} will hold since $*mynode_{p_1} = go_{p_2}$ by the FAS operation in the line, and this will result in I_{3d} holding since either $PC_{p_1} = 9$, or $PC_{p_1} \neq 9 \wedge *mynode_{p_1} = go_{p_2}$.

$\pi = p_i$ Here we have assumed $i > 2$, so I_{3eiv} and I_{3ei} once again show that the comparisoin will fail and go_{p_i} will be successfully placed into $*mynode_{p_{i-1}}$. Furthermore, PC_{p_i} will become 6. This proves that I_{3ev} continues to hold. No other part of I_3 is effected by the line execution.

line 6: By one execution of line 6, we mean that $*go_\pi$ will be read once. If *true*is returned by the read π will enter the CS, otherwise π will stay at line 6.

- I_1, I_2, I_4, and I_5 continue to hold as they are unaffected by this line.
- To prove I_3, we consider the cases $\pi = p_1$ and $\pi = p_i$, where $i > 1$.

$\pi = p_1$ By I_{3b} before the execution of the line, $*go_{p_1} = true$ so p_1 will enter the CS and be at line 7 after the execution.

I_{3a} holds since $prev_{p_1}$ is unaffected by the line.

I_{3b}, and I_{3c} hold since $PC_{p_1} = 7 \neq 9$ after the line.

I_{3d} holds since $k \geq 2 \wedge PC_{p_2} = 6$ implies that $*mynode_{p_1} = go_{p_2}$ by I_{3ev}.

I_{3e}, I_{3f}, and I_{3g} continue to hold as they are unaffected by this line execution.

$\pi = p_i$ Here we have assumed $i > 1$, so I_{3eii} states that the read will return *false*. Thus, the configuration is unchanged and I_3 continues to hold.

line 7: We remark that $\pi = p_1$ since all other processes in R are at lines $[5, 6]$ by I_{3eiii}.

- I_1, I_2, I_3, and I_4 continue to hold as they are unaffected by this line.
- We observe that I_{3f} states that TAIL $\in \mathbb{N}$, and I_{3a} states that $prev_{p_1} \notin \mathcal{N} \cup \{\texttt{nil}\}$. So we have shown that $\mathcal{A}' = \mathcal{A} - \{prev_{p_1}\}$, since $prev_{p_1}$ was freed in the executed line.

line 8: We remark that $\pi = p_1$ since all other processes in R are at lines $[5, 6]$ by I_{3eiii}.

- I_1, I_4 and I_5 continue to hold as they are unaffected by this line.
- We now split the proof into two cases, depending on whether p_1 is the only registered process.

$|R| = 1$ In this case I_{3f} shows that TAIL $= mynode_{p_1}$, and I_{3g} then implies that $*mynode_{p_1} =$ `nil` before the line execution. So, $*$TAIL will become non-nil and p_1 will leave the protocol making TAIL $\notin \mathcal{N} \cup \texttt{nil}$ by I_5 and $R' = \varnothing$. This shows that I_2 continues to hold.

I_3 continues to hold since it is now trivial.

$|R| > 1$ In this case, it is possible for the comparison to succeed, thus we further subdivide into two cases depending on the value of PC_{p_2}—there are only two cases due to I_{3eiii}.

$PC_{p_2} = 5$: By I_{3eiv} the comparison on line 8 will succeed, and $\pi = p_1$ will proceed to the remainder section. After the line execution the new p'_i is the old p_{i+1} for each i. In particular, p'_1 is the old p_2, and thus is on line 5 with $*prev_{p'_1} \neq \texttt{nil}$ and $prev_{p'_1} \notin \mathcal{N} \cup \{\texttt{nil}\}$. So I_{3a}, I_{3b}, and I_{3c} hold.

I_{3f} and I_{3g} continue to hold by the renaming.

If $|R'| = 1$, then the I_{3d} and I_{3e} hold trivially. Otherwise, I_{3d} holds since $PC_{p'_2} = 6 \implies *mynode_{p'_1} = go_{p'_2}$ by I_{3ev}. I_{3e} is unaffected for all $i > 2$. For $i = 2$, we notice that I_{3ev} continues to hold since the hypothesis was only strengthened, and the remaining subparts continue to hold since they were unaffected.

$PC_{p_2} = 6$: By I_{3ev} $next\_go_{p_1}$ will now hold go_{p_2}, and the comparison will succeed, leading to $PC_{p_1} = 9$. This proves I_{3c} will continue to hold. The remaining parts of I_3 are unaffected and thus continue to hold.

line 9: We remark that $\pi = p_1$ since all other processes in R are at lines $[5, 6]$ by I_{3eiii}. By I_{3c} we know that there are at least two process in R, and that $PC_{p_2} = 6$ and $next\_go_{p_1} = go_{p_2}$. So, the execution of the line makes: (i) p_1 leaves P and R, and $mynode_{p_1}$ leaves $\mathbb{N}$, (ii) The process $p'_i \in R'$ is the old $p_i \in R$, (iii) $*go_{p'_1}$ is now *true*. I_{3a} now holds since $prev_{p'_1} = mynode_{p_1}$ (by I_{3ei}) is no longer in $\mathbb{N}$. I_{3b} now holds since $PC_{p_1} = 6$ and $*go_{p'_1} = true$. I_{3c} holds trvially since $PC_{p_1} = 6 \neq 9$. I_{3f} and I_{3g} hold by the renumbering of processes in R'.

If $|R'| = 1$, then the I_{3d} and I_{3e} hold trivially. Otherwise, I_{3d} holds since $PC_{p'_2} = 6 \implies *mynode_{p'_1} = go_{p'_2}$ by I_{3ev}. I_{3e} is unaffected for all $i > 2$. For $i = 2$, we notice that I_{3ev} continues to hold since the hypothesis was only strengthened, and the remaining subparts continue to hold since they were unaffected.

□

4.2.3 Proof of Properties

The crisp invariant I identified in Figure 4-1 allows us to prove all of the properties of Algorithm 2 easily. Since the algorithm allocates and de-allocates shared memory nodes, we first demonstrate memory safety.

Lemma 4.2.2 (memory safety)**.** *A process never accesses a node or a go variable that has already been freed.*

Proof. For nodes: A process p only dereferences two shared nodes: its $mynode_p$ and its $prev_p$. The algorithm guarantees that $mynode_p$ is only dereferenced when $PC_p \in [2, 8]$, and $prev_p$ is only dereferenced when $PC_p = 5$. So, $mynode_p$ and $prev_p$ are in $\mathcal{A}$ when they are dereferenced, and I_5 thus guarantees that they are allocated and not yet freed when dereferenced.

For go variables: Process p itself does not act on go_p outside of line 3 and 6. The only other reference to a *go* variable is on line 9; but I_{3eiii} guarantees that p_1 is the only process that can be at line 9, and I_{3c} implies that $next\_go_{p_1} = go_{p_2}$ and p_2 in turn is on line 5 or 6 by I_{3eiii}. In particular, we have shown that go_p is only accessed when $PC_p \in [3, 6]$, strictly between its allocation (on line 1) and deallocation (on line 7). □

We now demonstrate the various safety and liveness properties of the algorithm in sequence.

Lemma 4.2.3 (mutual exclusion)**.** *There is at most one process p in the CS (with $PC_p = 7$) at a given time.*

Proof. A process p with $PC_p = 7$ is in the set R by definition. I_{3eiii} guarantees that all the p_i with $i \geq 2$ are not in the CS. So, p_1 is the only process that can be in the CS. □

Lemma 4.2.4 (FCFS)**.** *Algorithm 2 satisfies the FCFS property.*

Proof. I_{3f} and I_{3ei} together guarantee that the ordering $p_1, \ldots, p_k$ of process in R is unique. So, each process $q \in R$ has a unique identity $i \in [1, k]$ in each configuration such that $q = p_i$; we call this identity its *current position* in the process wait-queue. If process q completes line 4 before process q', the current position of q is smaller than the current position of q'. Since the relative order of current positions does not change for processes in the try section, the current position of q' is larger than the current position of q until q finishes the Critical Section and executes the Exit Section. □

Lemma 4.2.5 (starvation freedom)**.** *Algorithm 2 is starvation free.*

Proof. Assume that no process is in the CS or Exit and that at least one process is in the waiting room. By definition $|R| \geq 1$. I_{3b} guarantees that p_1 is either in the CS, the Exit, or will enter the CS after its next step. So, by our assumption, p_1 will eventually enter the CS if each process in the try section continues to take steps. This shows that under our assumption, as long as a process in the CS eventually leaves the Exit, another process will eventually enter the CS. Since Algorithm 2 satisfies FCFS, Starvation-Freedom follows. □

Lemma 4.2.6 (bounded exit)**.** *The Exit Section is bounded.*

Proof. The Exit Section finishes in at most two steps. □

4.2.4 RMR and Space Complexity

In this section, we analyze the efficiency of our main algorithm when it is used in a system with N processes. In order to express the tightest possible bounds and compare them to previous work we define $n = |P| \leq N$ as the total number of active processes.

Lemma 4.2.7. *Algorithm 2 is stateless and memory adaptive, i.e., it uses only $O(n)$ space.*

Proof. Since the algorithm requires no process to remember any variables in the Remainder Section, the algorithm is stateless. Since each process p frees go_p in the Exit Section, at most one such variable is allocated per active node. Invariant I_5 states that $\mathcal{A} = \mathcal{N} \cup \{prev_p | PC_p \in [5,7]\} \cup \{\text{TAIL}\}$ is the set of allocated nodes. We now use invariant I_{3ei} to see that all but one of the $prev_p$'s must be in the set $\mathbb{N}$, and use I_{3f} to see that TAIL $\in \mathbb{N}$ unless both $\mathbb{N}$ and $\{prev_p | PC_p \in [5,7]\}$ are empty. So, we see that in all cases $|\mathcal{A}| \leq n + 1$. □

We now show that RMR complexity of the algorithm is constant.

Lemma 4.2.8. *Algorithm 2 has RMR complexity $O(1)$ in both the DSM and CC model.*

Proof. The algorithm has only a single line that can be run more than once, namely line 6 which contains a **wait till** statement. In the DSM model, this line is reading a variable in p's partition of memory and thus costs zero RMRs. In the CC model, this line process p is repeatedly reading $*go_p$. Since, true is the only value ever written to $*go_p$ by another process, p can incur at most two RMRs on this line: the first to read it the first time, and if it is false, another to read it after it's modified to *true*. □

We summarize the results on Algorithm 2 in a theorem.

Theorem 4.2.9. *The main algorithm has the following properties: mutual exclusion, starvation freedom, FCFS, bounded exit, $O(1)$ RMR complexity in both DSM and CC models, $O(n)$ space complexity, space-frugality, and statelessness.*

4.3 Instantiations

In each attempt of the main algorithm, the attempting process p allocates its $mynode_p$ in the Try Section and frees its $prev_p$ in the Exit Section. The original algorithm is written for a shared memory machine with a linearizable memory allocator. We now show two simple ways to implement a memory allocator for this particular algorithm that lead to useful variations of the main algorithm.

4.3.1 Deriving a Node Toggling Algorithm

In order to prove that this is a valid implementation of a memory allocator for the presented algorithm, we must show that each time a node is being given out by **malloc**, it is a node that is not currently allocated. In order to show this, we define I_p^T, the *toggle invariant for process p*:

Algorithm 3 Our Node Toggle Algorithm Memory Allocator

For each process p:
$mynode_p[0], mynode_p[1]$: hold pointers to nodes
$t_p \in \{0, 1\}$: arbitrarily initialized

1: **malloc**$_p$(shared word)
2: $t_p \leftarrow 1 - t_p$
3: **return** $mynode_p[t_p]$
4: **free**$_p$(nodeptr) ▷ do nothing

$$I_p^T := \quad (p \in P \implies mynode_p = mynode_p[t_p]) \quad \wedge$$
$$(PC_p \in \{8, 9, 1\} \implies mynode_p[1 - t_p] \notin \mathcal{A})$$

We now prove the *toggle invariant* $I^T := \forall p, I_p^T$

Theorem 4.3.1. *I^T is an invariant of the toggle instantiation of the main algorithm.*

Proof. The proof is by induction on steps of the multiprocess system.

Base Case: Consider any process p in the initial configuration, we see that $p \notin P$ and $mynode_p[0], mynode_p[1] \notin \mathcal{A}$. So, I^T holds in the initial configuration.

Induction Step: Assume I^T holds in a given configuration, and consider what happens when some process π takes the next step.

- If π executes a line other than 1 and 7, I^T is unaffected and thus continues to hold after the step.
- If the next step of π is line 1, then $mynode_\pi[1 - t_\pi]$ is not allocated before the line execution by I^T. So, when t_π is complemented and $mynode_\pi[t_\pi]$ is returned, I_π^T continues to hold. I_p^T is unaffected for other processes $p \neq \pi$, so I^T continues to hold after the step.
- If the next step of π is line 7, then $\mathcal{A} = \mathbb{N}$ after the free operation, since I_{3g} shows that TAIL $\in \mathbb{N}$, I_{3ei} shows that $\{prev_p | PC_p \in [5, 7]\} \subset \mathbb{N}$. Since $\mathcal{N} = \{mynode_p | p \in P\}$ and $mynode_\pi = mynode_\pi[t_\pi]$ we conclude that $mynode_\pi[1 - t_\pi] \notin \mathcal{A}$.

□

Corollary 4.3.2. *The toggle allocator can be used with the main algorithm.*

Proof. I^T shows that nodes returned by malloc are never in $\mathcal{A}$, and I_5 shows that $\mathcal{A}$ is the set of allocated and not yet freed nodes. So, the toggle allocator always mallocs only unallocated nodes. □

Now that we have proved the correctness of the toggle allocator, we notice that we can actually inline the toggle allocation process to simplify the main algorithm to the *toggle algorithm*, presented below as Algorithm 4. In line 7 of this algorithm, NONNIL can be any value that is not equal to `nil`.

Algorithm 4 Our Node Toggle Algorithm: Derived by Instantiation from Our Main Algorithm

1: $t_p \leftarrow 1 - t_p$
2: $go_p \leftarrow \mathit{false}$
3: $\text{NODE}_p[t_p] \leftarrow$ `nil`
4: $prev_p \leftarrow \text{FAS}(\text{TAIL}, \&\text{NODE}_p[t_p])$ ▷ Doorway Ends
5: **if** $\text{FAS}(*prev_p, \&go_p) =$ `nil` **then**
6: **wait till** $go_p = \mathit{true}$
CRITICAL SECTION
7: **if** $(goptr_p \leftarrow \text{FAS}(\text{NODE}_p[t_p], \text{NONNIL})) \neq$ `nil` **then**
8: $*goptr_p \leftarrow \mathit{true}$

4.3.2 Deriving a Node Switching Algorithm

Algorithm 5 Our Node Switching Memory Allocator

For each process p: $previous_p$ is initially $\&\text{NODE}_p$

1: **malloc**$_p$(shared word)
2: **return** $previous_p$
3: **free**$_p$(nodeptr)
4: $previous_p \leftarrow$ nodeptr

We now show that the node switch allocator will only allocate unallocated nodes. To do this, we I_p^S, the *node switch invariant for process p*:

$$I_p^S \quad := \qquad p \in [2,7] \iff previous_p \in \mathcal{A}$$

We wish to show the *node switch invariant* $I^S := \forall p, I_p^S$

Theorem 4.3.3. *I^S is an invariant of the node switch instantiation of the main algorithm.*

Proof. The proof is by induction on steps of the multiprocess system.

Base Case: Consider any process p in the initial configuration, $PC_p = 1$ and $previous_p \notin \mathcal{A}$. So, I^S holds initially.

Induction Step: Assume I^S holds in a given configuration, and consider what happens when some process π takes the next step.

- If π executes a line other than 1 and 7, I^S continues to hold since no node is allocated or freed on the line and I_5 states that $\mathcal{A}$ will therefore be unaffected by the step.
- If the next step of π is line 1, by inductive hypothesis $previous_\pi \notin \mathcal{A}$ before line 1 and $previous_\pi = mynode_\pi \in \mathcal{A}$ after line 1. Therefore, I^S_π holds after the step. For $p \neq \pi, I^S_p$ is unaffected by the step, so I^S holds after the step.
- If the next step of π is line 7, by inductive hypothesis $prev_\pi \in \mathcal{A}$ before the step and $prev_\pi$ (which is also the new value of $previous_\pi$) is not in $\mathcal{A}$ after it is freed by I_5. So, I^S_π holds after the step. For $p \neq \pi, I^S_p$ is unaffected by the step, so I^S holds after the step.

□

Corollary 4.3.4. *The node switch allocator can be used with the main algorithm.*

Proof. I^S shows that nodes returned by malloc are never in $\mathcal{A}$, and I_5 shows that $\mathcal{A}$ is the set of allocated an not yet freed nodes. So, the node switch allocator always mallocs only unallocated nodes. □

Inlining the node switch allocator into the main algorithm yields the *node switch algorithm*, presented below as Algorithm 6. In line 7 of this algorithm, NONNIL can be any value that is not equal to `nil`.

Algorithm 6 Our Node Switching Algorithm: Derived by Instantiation from Our Main Algorithm

1: $mynode_p \leftarrow prev_p$
2: $*mynode_p \leftarrow \texttt{nil}$
3: $*go_p \leftarrow false$
4: $prev_p \leftarrow \text{FAS}(\text{TAIL}, mynode_p)$ ▷ Doorway Ends
5: **if** $\text{FAS}(*prev_p, go_p) = \texttt{nil}$ **then**
6: **wait till** $*go_p = true$

CRITICAL SECTION

7: **if** $(next\_go_p \leftarrow \text{FAS}(*mynode_p, \text{NONNIL})) \neq \texttt{nil}$ **then**
8: $*next\_go_p \leftarrow true$

Chapter 5

Abortable Mutual Exclusion

5.1 Introduction

The *Mutual exclusion problem*, proposed by Dijkstra, is an abstraction of a *lock* that is owned by at most one process at any time [52]. The *abortable* mutual exclusion problem, proposed by Scott and Scherer [180] in response to the needs in real time systems and databases, is the variant that allows processes to abort from their attempt to acquire the lock. Worst-case constant remote memory reference (RMR) algorithms for mutual exclusion using hardware instructions such as Fetch-and-Add or Fetch-and-Store have long existed for both Cache Coherent (CC) and Distributed Shared Memory (DSM) multiprocessors [13, 83, 152, 41, 148, 10, 56], but no such algorithms are known for abortable mutual exclusion. Even relaxing the worst-case requirement to amortized, algorithms are only known for the CC model—Lee's deterministic algorithm [138] and Giakkoupis and Woelfel's randomized algorithm [72]. In this chapter, we improve this state-of-the-art by designing a deterministic algorithm that uses Fetch-and-Store (FAS) to achieve amortized constant RMR in both the CC and DSM models. A further highlight of our algorithm is that a process can abort in $O(1)$ steps in the worst-case, a property that neither Lee's nor Giakkoupis and Woelfel's algorithm has. In the rest of this section, we specify the problem, the RMR complexity metric, and describe our contribution in the context of prior research.

5.1.1 Specification of Abortable Mutual Exclusion

In the abortable mutual exclusion problem, each process is modeled by five sections of code—Remainder, Try, Critical, Exit, and Abort sections. A process stays in the Remainder section when

it does not need the lock and, once it wants the lock, it executes the Try section concurrently with others that are also competing for the lock. Anytime a process is outside the Remainder section, the environment can send an "abort" signal to the process (how the environment sends this signal to a process does not concern us). From the Try section, a process jumps either to the Critical Section (CS) or to the Abort section, with the proviso that a process may jump to the Abort section only if it receives the "abort" signal from the environment while in the Try section. While in the CS, a process has exclusive ownership of the lock. When it no longer needs the lock, the process gives it up by executing the Exit section to completion and then moving back to the Remainder section. If a process enters the Abort section from the Try section, upon completing the Abort section the process moves back to the Remainder section.

The *abortable mutual exclusion problem* consists of designing the code for the Try, Exit, and Abort sections so that the following properties are satisfied [104].

- Mutual Exclusion: At most one process is in the CS at any time.
- Bounded Exit: There is a bound b such that each process in the Exit section completes that section in at most b of its own steps.
- Bounded Abort: There is a bound b such that, once a process receives the abort signal from the environment, it will enter the CS or the Remainder section in at most b of its own steps. We call this bound b the *abort-time*.
- Starvation-Freedom: Under the assumption that no process stays in the CS forever and no process stops taking steps while in the Try, Exit, or Abort sections, if a process in the Try section does not abort, it subsequently enters the CS.

We now state another desirable property that has never been proposed or investigated earlier. In any application, the environment sends the abort signal to a process only when there is some urgent task that the environment needs the process to attend to. In such a situation, we would want the process to "quickly" abort from its attempt to acquire the lock. Accordingly, we define *abort-time* as the worst-case number of steps that a process takes between receiving an abort signal and subsequently entering the CS or the Remainder section. So, we would like the abort time to be a constant independent of the number of processes:

- Fast Abort: The abort-time is an absolute constant.

Next, we define a novel fairness property called *Airline First Come First Served* (AFCFS), which is a natural relaxation of the standard First Come First Served (FCFS) property for the abortable setting. For intuition, imagine you are waiting to check-in in a long airline queue. You will not mind if someone who was ahead of you but left for the restroom (i.e. aborted) comes back to their old position in the queue. However, you will not allow a newcomer to take a position ahead of you in the queue. Before formalizing this property below, recall that FCFS for standard mutual exclusion (i.e. without the abort feature) is formalized using either the notion of a bounded Doorway [135], or more directly by the following condition: there is a bound b such that if a process p_1 performs b steps of the Try section before a process p_2 enters the Try section, then p_2 does not enter the CS before p_1.

Turning our attention back to the abortable problem, we say a process begins a *passage* when it first enters the Try section or when it first enters the Try section after the end of its previous passage, and the passage ends when the process completes the Exit section. Note that a process can abort arbitrarily many times within each of its passages. Let π_1 be a passage of p_1 and π_2 a passage of p_2. We say that π_1 *b-precedes* π_2 if p_1 executes b steps $s_1, s_2, \ldots, s_b$ of the Try section in π_1 before the start of π_2 and p_1 does not abort in π_1 after step s_1.

- Airline First Come First Served (AFCFS): There is a bound b such that if a passage π_1 by process p_1 b-precedes a passage π_2 by process p_2, then p_2 does not enter the CS in π_2 before p_1 enters the CS in π_1.

Note that if no aborts are ever performed, then AFCFS coincides identically with standard FCFS.

5.1.2 RMR Complexity

In a *cache-coherent* (CC) machine each process has a cache. A read operation by a process p on a shared variable X fetches a copy of X from shared memory to p's cache, if a copy is not already present. Any non-read operation on X by any process invalidates copies of X at all caches. An operation on X by p counts as a *remote memory reference* (RMR) if either the operation is not a read or X's copy is not in p's cache. In a *distributed shared memory* (DSM) machine, instead of caches, shared memory is partitioned, with one partition residing at each process, and each shared variable resides in exactly one partition. Any operation (read or non-read) by a process p on a shared variable X is counted as an RMR if X is not in p's partition.

The *worst-case RMR complexity* of an algorithm is the maximum number of RMRs incurred by a process between the times of leaving and subsequently re-entering the Remainder section, not counting any RMRs incurred in the CS. The *amortized RMR complexity* of an algorithm is the maximum value of x/y, where x is the total number of RMRs performed in the Try, Abort, and Exit sections by all processes and y is the total number of times the Try section was entered by any process. In general, the RMR complexity is a function of n, which is the maximum number of processes that execute the algorithm.

5.1.3 The Impact of Synchronization Primitives on RMR Complexity

The RMR complexity of the mutual exclusion problem is well known to depend on the synchronization primitives available for use in algorithms. A long standing lower bound due to Cypher [42], and an even stronger lower bound due to Attiya, Hendler, and Woelfel [17] establish that constant RMR complexity is unachievable even for standard mutual exclusion using read, write, and Compare-and-Swap (CAS). In fact, sub-logarithmic deterministic RMR complexity is unachievable even with amortization using only read, write, and CAS [17]. On the other hand, it has long been known that standard mutual exclusion has constant RMR algorithms using non-conditional primitives, such as Fetch-and-Add (F&A) [13, 10] or Fetch-and-Store (FAS) [83, 152, 41, 148, 56, 108] (the constant bound holds only on CC machines for some of these algorithms, and for both CC and DSM machines for the others). Thus, FAS or F&A can be more effective than CAS when solving mutual exclusion and related problems. Our algorithm is based on the FAS primitive.

There is another sense in which FAS and F&A are stronger than CAS: unlike FAS and F&A, CAS admits a deterministic $O(1)$ RMR implementation using read and write operations [79]. Consequently, any algorithm A that uses read, write, and CAS can be transformed into an algorithm A' that uses only read and write by replacing each instance of CAS in A with its implementation via reads and writes. The resulting algorithm A' would have the same asymptotic RMR complexity as A, and may appear to solve the same problem as A does. However, the latter is not always the case because the CAS implementation via reads and writes is blocking. As a result, if A is a mutual exclusion algorithm that uses CAS in the Exit section, A' may not satisfy the Bounded Exit property even if A does. If A is an *abortable* mutual exclusion algorithm that uses CAS in its Try section, A' no longer solves the problem because the CAS implementation via reads and writes is not abortable.

5.1.4 Our Contribution

We investigate whether $O(1)$ RMR complexity is feasible for *abortable* mutual exclusion in the CC and DSM models. If the goal is $O(1)$ *worst case* RMR complexity, the problem has been and continues to be open. On the other hand, for $O(1)$ *amortized* RMR complexity, there has been some progress, albeit only for the CC model. In light of the lower bound described above, it follows that any algorithm that aspires for $O(1)$ amortized RMR complexity must either use synchronization primitives other than CAS or use randomization. The former approach was taken by Lee [138], and the latter by Giakkoupis and Woelfel [72]. Lee's algorithm (the second one in his thesis [138]) is deterministic and uses FAS and CAS, but its amortized RMR complexity of $O(1)$ was unproven and even unobserved. Giakkoupis and Woelfel's algorithm, uses CAS and randomization, but has expected $O(1)$ amortized RMR complexity only against an oblivious scheduler. Their algorithm uses infinite arrays but they also sketch how to keep the memory use bounded to a polynomial (of unknown constant degree).

For the DSM model, there are no algorithms using common synchronization primitives of even sub-logarithmic amortized RMR complexity, let alone $O(1)$. In fact, only two algorithms of bounded RMR complexity are known: Jayanti's algorithm [104] and the first algorithm in Lee's thesis [138], both of which have logarithmic worst-case RMR complexity.

In this work, we design a deterministic algorithm that uses FAS and guarantees $O(1)$ amortized RMR complexity for both the CC and DSM models.[1] Our algorithm supports Fast Abort: a process aborts within 6 steps of receiving the abort signal. The algorithms of Lee and Giakkoupis and Woelfel do not have the fast abort property, making ours the first to support Fast Abort and have $O(1)$ amortized RMR complexity in either model. (To the familiar reader, Lee's algorithm does not satisfy fast abort because an aborting process p may find as many as $\Theta(n)$ aborted nodes ahead of its node. In this case, p removes each of these nodes from the wait-queue, taking $\Theta(n)$ steps, before aborting.)

Our algorithm has the following additional desirable properties: It satisfies AFCFS, requires only $O(1)$ space per active or aborted process, and supports an arbitrary number of processes of arbitrary names. It is also succinct with fewer than a dozen lines of code.

Our algorithm is inspired by a long line of research on *queue locks*, especially of the MCS variety [152], where waiting processes are organized as a linked list. The MCS lock uses both FAS and

[1] The Fetch-and-Store (FAS) instruction is a read-modify-write instruction that behaves as follows: FAS(X, v) changes shared variable X's value to v and returns X's previous value.

CAS, and does not satisfy Bounded Exit. Craig refines this algorithm to use only FAS and bound the exit section [41]. A further simplification of Craig's algorithm is presented by Jayanti et. al. [108]. These locks use only $O(1)$ space per active process and support an arbitrary number of processes which Lee's algorithm as well as ours inherit. The second algorithm in Lee's Thesis [138] builds on Craig's algorithm to implement the abort feature on CC machines. Lee's clever insight is that an aborting process could leave its node in the queue and try to reclaim that position the next time it attempts to capture the lock (thereby realizing AFCFS). Our algorithm builds on those of Jayanti et. al. and Lee with two further insights. The first concerns what actions a process takes upon receiving the abort signal. In Lee's algorithm, an aborting process—even as it leaves its own node in the queue—kicks out all contiguous aborted nodes in front of it. In contrast, our algorithm kicks out at most one node. This idea makes the proof of starvation freedom more complex, but is crucial to achieving Fast Abort. Second, our algorithm incorporates the requisite indirection to ensure constant RMR complexity even on DSM machines. This indirection is algorithmically subtle because, even though it "falsely" wakes waiting processes multiple times, neither correctness nor the constant amortized bound is broken. We prove mutual exclusion through invariants, and starvation-freedom and complexity analysis through potential functions.

5.1.5 Prior Research

Algorithm	**Primitive**	**RMRs**	**WC / Amrt**	**Det.**	**Space**	**DSM**	**Fairness**	**Fast Abort**
Scott et al. [180]	FAS, CAS	∞	WC	✓	∞	✓		
Scott CLH-NB [179]	FAS, CAS	∞	WC	✓	∞	✓		
Scott MCS-NB [179]	FAS, CAS	∞	WC	✓	∞	✓		
Jayanti [104]	CAS	$\Theta(\log n)$	WC	✓	$\Theta(n)$	✓	FCFS	
Lee Alg 1 [138]	None	$\Theta(\log n)$	WC	✓	$\Theta(n \log n)$	✓		
Lee Alg 2 [138]	FAS, CAS	$\Theta(n)/\Theta(1)$	WC/Amrt	✓	$\Theta(n)$		AFCFS	
Lee Alg 3 [138]	FAS	$\Theta(n^2)$	WC	✓	∞		FCFS	
Lee Alg 4 [138, 137]	FAS	$\Theta(n^2)$	WC	✓	$\Theta(n^2)$		FCFS	
Woelfel et al. [169]	CAS	$O(\frac{\log n}{\log \log n})$	WC		$\Theta(n)$			
Giakkoupis et al. [72]	CAS	$\Theta(1)$	Amrt		∞			
Alon et al. [7]	F&A, CAS	$O(\frac{\log n}{\log \log n})$	WC	✓	$O(n^2)$			
This Chapter Alg 7	FAS	$\Theta(1)$	Amrt	✓	$\Theta(n)$		AFCFS	✓
This Chapter Alg 8	FAS	$\Theta(1)$	Amrt	✓	$\Theta(n)$	✓	AFCFS	✓

Table 5.1: Summary of abortable locks. The columns describe: RMR complexity; whether the complexity is worst case (WC) or amortized (Amrt); whether the algorithm is Deterministic (Det.) or randomized; space complexity; whether the RMR bound holds for the DSM model; what fairness condition (if any) the algorithm satisfies; and whether the algorithm supports Fast Abort.

We list previous algorithms for abortable mutual exclusion and their properties in Table 5.1.

Scott and Scherer were the first to formulate the abortable mutual exclusion problem [180], but their algorithm and Scott's subsequent algorithm [179] have unbounded RMR complexity. Jayanti presented the first algorithm of bounded RMR complexity [104]. His algorithm uses CAS and has $O(\log n)$ RMR complexity which is optimal by the aforementioned lower bound [17]. Giakkoupis and Woelfel gave a CAS-based randomized abortable lock of $O(1)$ expected amortized RMR complexity [72], which contrasts positively with the lower bound [17] that deterministic sub-logarithmic complexity is infeasible even with amortization. Another CAS-based algorithm by Woelfel and Pareek [169] uses randomization yet again to beat the lower bound and attain a sub-logarithmic, albeit non-constant, expected RMR complexity.

Lee's dissertation has four FAS based deterministic algorithms with worst-case RMR complexities of $\Theta(\log n)$, $\Theta(n)$, $\Theta(n^2)$, and $\Theta(n^2)$ [138]. The second algorithm has $O(1)$ *amortized* RMR complexity and satisfies AFCFS, although neither of these facts is proved by him. Alon and Morrison [7] designed a deterministic Fetch-and-Add based algorithm with sub-logarithmic, but non-constant RMR complexity.

Of the above, only two algorithms have bounded RMR complexity in the DSM model—Jayanti's [104] and Lee's first algorithm [138].

5.2 Line Numbering Convention

We adopt the standard model in which an execution of an algorithm is a sequence of atomic steps, where a step consists of any one process performing an instruction on a shared variable and a bounded number of instructions on its local variables. Accordingly, we label each shared memory instruction in an algorithm with a distinct line number, and a step consists of performing that shared memory instruction as well as all the subsequent instructions on local variables, up to and excluding the next numbered shared memory instruction.

5.3 An O(1) Algorithm for CC

In this section, we describe a very succinct algorithm that is designed to have the $O(1)$ Amortized RMR complexity guarantee only on CC machines. Our algorithm uses ideas that are similar to Lee's [138], but it is simpler and it uses only the FAS synchronization primitive; additionally, we furnish it with a proof of correctness via invariants and an amortized complexity analysis.

Algorithm 7 : Amortized constant RMR abortable lock for CC machines. Code shown for process p. Process p jumps to the Abort Section if the abort signal is on and p is at Line 4 or 5.

Variables

- A *node* is a single shared memory word that can hold a pointer, or `nil`, or `token`.
- TAIL: shared variable that points to a node. Initially, it points to a node that is allocated and initialized to `token`.
- When a process p first participates in the algorithm:
  - It allocates two local variables $mynode_p$ and $pred_p$ both of which point to the same freshly allocated node initialized to `nil`.
  - It allocates an uninitialized local variable v_p that can hold a pointer, `nil`, or `token`.

Section TRY(p)

1: **if** FAS($*mynode_p$, `nil`) $\neq pred_p$ **then**
2: $pred_p \leftarrow$ FAS(TAIL, $mynode_p$)
3: $v_p \leftarrow$ FAS($*pred_p$, `nil`)
if $v_p \notin$ {`nil`, `token`} **then** $pred_p \leftarrow v_p$
while $v_p \neq$ `token` **do**
4: **if** $(v_p \leftarrow *pred_p) \notin$ {`nil`, `token`} **then**
5: $v_p \leftarrow$ FAS($*pred_p$, `nil`)
if $v_p \notin$ {`nil`, `token`} **then** $pred_p \leftarrow v_p$

Section EXIT(p)

6: $*mynode_p \leftarrow$ `token`
$mynode_p \leftarrow pred_p$

Section ABORT(p)

7: $*mynode_p \leftarrow pred_p$

Remark 5.3.1. *The algorithm only requires that* `token` *is different from every value that* $mynode_p$ *could take on; so we can use the address* &TAIL *as* `token`*.*

5.3.1 Informal Description

We present our abortable lock as Algorithm 7. To build intuition about the algorithm, we start by describing how the lock would work if processes do not abort. Then, we describe the parts of the code that facilitate efficient aborting.

Fundamentally, our lock is a *queue lock*, i.e. all k processes that have finished the doorway but not yet the Exit section form a process wait-queue, $Q = q_1, \ldots, q_k$. We call the first process in the queue, q_1, the *head process* and the last process q_k the *tail process*. Furthermore, for each process q_i, the process q_{i-1} is its *predecessor* and q_{i+1} is its *successor*—apart from the head process which has no predecessor and the tail process which has no successor.

In the algorithm, the wait-queue is represented as follows. Each process p has two associated local variables $mynode_p$ and $pred_p$, which stands for predecessor of p. Each of these variables holds the address of a node—a single word in shared memory—that stores either `nil`, the address of another node, or the special value `token`. The abstract wait-queue $q_1, \ldots, q_k$ is represented by the list $mynode_{q_1}, \ldots, mynode_{q_k}$; this list is linked by the predecessor pointers, with $pred_{q_i}$ pointing to the same node that $mynode_{q_{i-1}}$ points to, for each $i \in [2, k]$. Furthermore, $pred_{q_1}$ points to a node that is not any process p's $mynode_p$; we call this node the *anchor node* and let a_0 denote its address. For brevity, we let a_i denote $mynode_{q_i}$, and define the *node wait-queue* to be the linked list of node addresses $a_0, a_1, \ldots, a_k$. In the remainder of the description, we use the term *wait-queue* to refer to either the process or node wait-queue as disambiguated by context. The final important component of the queue representation is the shared variable TAIL which holds the address a_k of the tail node. When a process that is not currently in the wait-queue wishes to join, it simply needs to apply an atomic FAS on TAIL to secure its predecessor node's address and simultaneously insert its node as the new tail of the wait-queue.

Each process uses its node to signal to its successor whether or not it has permission to enter the CS. In particular, when q_i leaves the CS, it replaces the `nil` in its node a_i with `token`, thereby signaling its successor q_{i+1} that it may enter the CS. Accordingly, q_{i+1} simply spins on the shared variable $*pred_{q_{i+1}}$, and enters the CS only after it finds `token` there. The algorithm ensures that $*a_0 =$ `token` and $*a_i =$ `nil` for $i \in [1, k]$ so that the head process is in the CS while all others wait.

In an execution of the Try Section, a process p performs the doorway by swapping `nil` into $mynode_p$ to initialize (Line 1), and joining the queue by swapping the address $mynode_p$ with TAIL and storing the result in $pred_p$ (Line 2). Then p spins, repeatedly reading $*pred_p$ into v_p (Line 4) until it becomes `token` (the condition of the while loop right before Line 4), at which point p proceeds to the CS. We ignore Line 3 for now, until we get to describing aborting.

Our protocol ensures that the anchor node a_0 holds `token`, which implies that q_1 is either already in the CS or enabled to enter the CS[2]. As q_1 exits, it enables its successor by writing `token` to $*a_1$ at Line 6. This ties up a_1 since its successor needs to read the `token` inside it in order to acquire the CS; however, it frees up the old anchor node a_0 which was the node that q_1 got the `token` from when it entered the CS. So, q_1 relinquishes its ownership of a_1 to let it become the new anchor node, and claims ownership of a_0 as its new *mynode*; this node assignment also happens on

[2]When we say process p is *enabled* to enter the CS, we mean that there is a bound b such that p will enter the CS if it takes b steps, regardless of how its steps interleave with the steps of other processes.

the same line, Line 6, since it is a local action.

We now describe aborting. In our algorithm, even if a process p receives the abort signal from the environment at or before Line 3, it is allowed to jump to the Abort section only after completing Line 3 (which, as already highlighted in Section 5.2, includes completing all local actions associated with Line 3). It follows that any aborting process must be in the wait-queue at Line 4 or Line 5.

When a process q_i in the wait-queue wishes to abort, it simply replaces the `nil` in its node a_i with the pointer $pred_{q_i}$ (Line 7), and moves to the Remainder Section. However, the node a_i that q_i owns is still a part of the linked list, so we continue to regard a_i as a member of the wait-queue. In fact, if q_i executes the Try Section immediately after aborting, it will notice that $*mynode_{q_i} = pred_{q_i}$ in the comparison at Line 1, and thereby skip Line 2 and reclaim its old position in the wait-queue. On the other hand, if q_i continues to stay in the Remainder section, process q_{i+1}, which is continually reading the value of $*pred_{q_{i+1}}$ (i.e. $*a_i$) at Line 4, will notice that $*a_i$ has a non-`nil`, non-`token` value. Thus, q_{i+1} will perform the FAS at Line 5 that splices a_i out of the linked list, by simultaneously setting $pred_{q_{i+1}}$ to a_{i-1} and $*a_i$ to `nil`. If q_i attempts the Try Section now, it will immediately discover that it has been removed from the queue, since `nil` has replaced $pred_{q_i}$ in $*mynode_{q_i}$ (Line 1).

Next we turn our attention to Line 3, the only line that we have not described yet. Notice that this line is identical to Line 5, yet, its placement at Line 3 is pivotal to ensuring Starvation Freedom. To see this, consider a scenario where processes q_1 and q_2 are both at Line 4 of the Try Section, and the following sequence of two actions repeats an unbounded number of times:

1. The anchor node a_0 has `token`. However, rather than finding this token and entering the CS, q_1 aborts by executing Line 7. As soon as it aborts, q_1 executes Lines 1 and 2 of the Try Section and regains its position at the front of the wait-queue. If Line 3 were not there, q_1 would find itself back at Line 4 of the algorithm.

2. q_2 executes Line 4. Since $*mynode_{q_1} =$ `nil`, the condition of the if-statement and while-loop both fail and q_2 remains at Line 4.

In the above scenario, neither process ever stops taking steps and q_2 does not abort, yet q_2 never enters the CS, thereby violating Starvation Freedom.

The essence of the problem is that process q_1 is enabled to get into the CS every time it enters the Try Section, but it is never taking enough steps to reach the CS before aborting. Line 3 alleviates this problem by forcing every process to examine the node in front of it each time it

enters the Try Section. Thus, the process is forced to make progress by entering the CS when it is enabled, or by splicing out its predecessor node if it is aborted. In our analysis, we give a rigorous argument by potential function that this small modification yields an algorithm that satisfies Starvation Freedom.

The following theorem summarizes the properties of Algorithm 7.

Theorem 5.3.2. *Algorithm 7 solves Abortable Mutual Exclusion for an unbounded number of processes of arbitrary names using the FAS synchronization primitive. In particular, the algorithm satisfies Mutual Exclusion, Bounded Exit ($O(1)$ steps), Fast Abort, Starvation-Freedom, and AFCFS. It uses only $O(1)$ space per process, and in the CC model has $O(1)$ amortized RMR complexity.*

In the next section, we prove this theorem via an invariant which specifies precisely, what values the variables of the algorithm take on and when, including when each node takes on each of the three values—`nil`, `token`, or pointer to another node.

5.4 Correctness and Efficiency of the CC Algorithm

We define P to be the set of processes, and N to be the set of $|P| + 1$ nodes that are allocated initially, as described in the "Variables" section of Algorithm 7.

Theorem 5.4.1. $I \triangleq \bigwedge_{j=1}^{13} I_j$ *is an invariant of Algorithm 7.*

Proof. We will prove the invariant by induction on steps of the multiprocess system. In particular, we consider what happens when a process π executes its next step.

Base Case: At the beginning $k = 0$ and a_0 is the node pointed to by TAIL. This initialization ensures that $a_0 \in N$ and $a_0 \neq mynode_p$ for each $p \in P.I_1$, and I_4 hold true since TAIL $= a_0$. I_2, I_3, I_8, and I_{12} hold trivially since $k = 0$. I_5 holds since $pred_p = mynode_p \in N$ for every $p \in P$. I_6, and I_{10} hold since $*mynode_p =$ `nil` $\neq pred_p$ for every $p \in P$ and since $Q = \varnothing$. I_7, I_8, I_9, I_{12}, and I_{13} hold trivially since $PC_p = 1$ for every $p \in P$. I_{11} holds since $*a_0$ is initialized to `token`.

Induction Step: We assume that invariant holds in a particular configuration, and consider what happens if the next step is taken by some process π executing one of the six possible lines.

There is an integer $k \geq 0$, and a sequence $A = a_0, \ldots, a_k$ of $k+1$ distinct nodes and a sequence $Q = q_1, \ldots, q_k$ of k distinct processes such that

$I_1)$ $\text{TAIL} = a_k$

$I_2)$ $\forall i \in [1,k], mynode_{q_i} = a_i$

$I_3)$ $\forall i \in [1,k], pred_{q_i} = a_{i-1}$

$I_4)$ $N = \{a_0\} \cup \{mynode_p \mid p \in P\}$

$I_5)$ $\forall p \in P,\ pred_p \in N$

$I_6)$ $\forall p \in P, (PC_p = 1) \implies (*mynode_p = pred_p \iff p \in Q)$

$I_7)$ $\forall p \in P, (PC_p \neq 1) \implies (*mynode_p = \texttt{nil})$

$I_8)$ $\forall p \in P, (PC_p = 2) \implies (p \notin Q)$

$I_9)$ $\forall p \in P, (PC_p \notin \{1,2\}) \implies (p \in Q)$

$I_{10})$ $\forall p \in P, *mynode_p \in \{\texttt{nil}, \texttt{token}, pred_p\}$

$I_{11})$ $((k = 0) \vee (k \geq 1 \wedge PC_{q_1} \neq 6)) \implies (*a_0 = \texttt{token})$

$I_{12})$ $(k \geq 1 \wedge PC_{q_1} = 6) \implies (*a_0 \in \{\texttt{nil}, \texttt{token}\})$

$I_{13})$ $\forall p \in P, p \neq q_1 \implies PC_p \neq 6$

Note: by I_1, I_2, I_3, and I_4, the queue of nodes starts with the tail and ends with the unique node a_0 that is no process p's $mynode_p$. This immediately implies that k and the sequences A and Q are unique.

Figure 5-1: Invariant of Algorithm 7.

line 1: We consider two cases: $*mynode_\pi = pred_p$ and $*mynode_\pi \neq pred_p$.

If $*mynode_\pi = pred_p$, then $\pi = q_i$ for some $i \in [1,k]$ by I_6. So, the comparison on line 1 will fail and PC'_π will become 3. The swap on line 1 will guarantee I_7 and I_{10} after the execution and π will continue to be q_i so I_9 will continue to hold. The rest of the invariant continues to hold trivially since it is unaffected by the execution.

If $*mynode_\pi \neq pred_p$, then $\pi \neq q_i$ for any $i \in [1,k]$ by I_6. So, the comparison on line 1 will succeed and PC'_π will become 2. The swap on line 1 will guarantee I_7 and I_{10} after the execution and π will continue to not be any of the q_i, and therefore I_8 will continue to hold. The rest of the invariant continues to hold trivially since it is unaffected by the execution.

line 2: Due to the execution of line 2, the value k' will become $k+1$, $q_1, \ldots, q_k$ will

remain the same and π will become $q_{k'}$ and $mynode_\pi$ will become $a_{k'}$, and $PC'_\pi = \{3\}$. π was not in Q before the step by I_8, and and $mynode_\pi$ was not in A before the step since I_2 and I_4 guarantee that the $k+1$ nodes that were in A before were either $mynode_q$ of some $q \in Q$ or $a_0 \notin \{mynode_p \mid p \in P\}$. I_1 will continue to hold due to the FAS on line 2, I_2 and I_3 will hold for $i = k'$ also due to the FAS (and will continue to hold for the other values of i as they are unaffected). I_9 will hold since $\pi = p_{k'}$. I_{12} will continue hold since $PC'_\pi \neq 6$. The remaining invariants will continue to hold since they are unaffected.

line 3: I_9 shows us that $\pi = q_i$ for some $i \in [1, k]$. So, $pred_\pi = a_{i-1}$ by I_3. We consider three cases: $*a_{i-1} = \texttt{nil}$, $*a_{i-1} = \texttt{token}$, and $*a_{i-1} = pred_{q_{i-1}}$. ($I_{10}$ guarantees that this is a complete list of cases.) I_{10} continues to hold in all cases since `nil` was the new value written into the node.

If $*a_{i-1} = \texttt{nil}$, the if-condition fails and the while-condition fails and π goes to line 4. All invariants are unaffected and therefore continue to hold.

If $*a_{i-1} = \texttt{token}$, we can infer that $i = 1$, since I_6 and I_7 guarantee that every a_j with $j > 1$ holds either `nil` or $pred_{p_j}$, and $pred_{p_j} \neq \texttt{token}$ by I_5. We observe that the if-condition fails and the while-condition succeeds and $\pi = p_1$ goes into the CS. $*a_0 = \texttt{nil}$, so I_{12} continues to hold. PC'_π becomes 6, but $\pi = q_1$, so I_{13} continues to hold. The remaining invariants are unaffected and thus continue to hold.

If $*a_{i-1} = pred_{q_{i-1}}$, then $*a_{i-1} \notin \{\texttt{nil}, \texttt{token}\}$. So, the if-condition succeeds, $pred'_\pi$ becomes $pred_{q_{i-1}}$, the while-condition succeeds, and PC'_π becomes 4. Effectively, the old q_{i-1} has been removed from the wait-queue Q, so k becomes $k-1$; $q_i, \ldots, q_k$ become $q'_{i-1}, \ldots, q'_{k-1}$ and $a_i, \ldots, a_k$ become $a'_{i-1}, \ldots, a'_{k-1}$. ($q_1, \ldots, q_{i-2}$ and $a_0, \ldots, a_{i-2}$ remain the same; note that I_{11} and I_{12} imply that $i \geq 2$.) I_3 continues to hold since $pred_{q'_{i-1}} = pred_{q_{i-1}} = a_{i-2} = a'_{i-2}$. The old q_{i-1} was removed from the Q, however the contrapositive of I_7 implies that $PC_{q_{i-1}} = 1$, and thus since $*mynode'_{q_{i-1}} = \texttt{nil}$, I_6 and I_9 continue to hold. The remaining invariants continue to hold since they are unaffected.

line 4: I_9 shows us that $\pi = q_i$ for some $i \in [1, k]$. So, $pred_\pi = a_{i-1}$ by I_3. We consider three cases: $*a_{i-1} = \texttt{nil}$, $*a_{i-1} = \texttt{token}$, and $*a_{i-1} = pred_{q_{i-1}}$. ($I_{10}$ guarantees that this is a complete list of cases.)

If $*a_{i-1} \in \{\texttt{nil}, \texttt{token}\}$, the same arguments as for line 3 apply and the invariant

continues to hold.

If $*a_{i-1} = pred_{q_{i-1}}$, the if-condition fails, and $PC'_\pi = 5$, the invariant is unaffected and thus continues to hold.

line 5: The argument is identical to that for line 3.

line 6: Since $PC_\pi = 6$, I_{13} implies that $\pi = q_1$. Line 6 therefore removes q_1 from Q, thus $k' = k - 1$. $q_2, \ldots, q_k$ become $q'_1, \ldots, q'_{k'}$, and $a_1, \ldots, a_k$ become $a'_0, \ldots, a'_{k'}$. a_0 becomes $mynode_\pi$, and therefore, $a_1 = a'_0 \notin \{mynode'_p \mid p \in P\}$. I_1, I_2, and I_3 continue to hold since they are simply translated to $i' = i - 1$. I_4 continues to hold since the old nodes are the same as the new nodes, just $mynode_\pi$ become a'_0 and a_0 became $mynode'_\pi$. Since, $*a_0 \in \{\texttt{nil}, \texttt{token}\}$, we see that $*mynode'_\pi \neq pred_\pi \in N$ and I_6 and I_{10} continue to hold. I_9 continues to hold since $PC'_\pi = 1$. $*a'_0 = \texttt{token}$, so I_{11}, I_{12} hold. The remainder of the invariant is unaffected and thus continues to hold.

line 7: Since $PC_\pi > 2$, I_9 implies that $\pi \in Q$. So, let $i \in [1, k]$, such that $\pi = q_i$. $PC'_\pi = 1$, but $*mynode'_\pi = pred_\pi$ so I_6 and I_{10} continue to hold. The remainder of the invariant is unaffected and thus continues to hold.

□

Corollary 5.4.2. *Algorithm 7 satisfies mutual exclusion.*

Proof. I_{13} states that only process q_1 can be in the critical section. □

Theorem 5.4.3. *Algorithm 7 satisfies AFCFS where the doorway constitutes Lines 1 and 2 of the Try Section.*

Proof. In order to show this property, we prove the following stronger statement by induction: *if process $p = q_i$ in the queue and process $p' = q_j$ in the queue with $j > i$, then p started its passage before p' finished its doorway.* This statement is true initially since Q is empty. The statement continues to hold inductively whenever any process leaves Q from any position. Thus, we are left with the case where a new process p enters Q. Since this case occurs only if p executes Line 2, and thereby just finishes the doorway, it is clear that every other process in Q has started its passage before p finished its doorway in its current passage. Now in conjunction with our inductive statement, we observe that by I_{13}, only the first process in Q, q_1 , can be in the CS; that finishes the proof. □

In order to show starvation freedom, we will define a distance δ_q for each process q that is in Q but not yet in the Exit Section. δ_q represents how far process q is from reaching the Critical Section, and it will attain its minimum value, one, if and only if q is in the Critical Section. To show that q will eventually reach the CS, we will prove that δ_q will decrease if q is not already in the CS by demonstrating: (1) δ_q is non-increasing; (2) There is a non-empty set $C(q) \in P$ of *critical processes* with respect to q, such that if any $q' \in C(q)$ takes a step δ_q will decrease, and if δ_q does not decrease in the next step, then the set of critical processes will remain unchanged. Thus, since every process in the try section must eventually take a step, δ_q will eventually decrease to one (if q does not abort) at which point point q will reach the Critical Section.

Before defining δ_q, we define the function $f(pc)$ that maps a program counter value $pc \in \{1, 3, 4, 5, 6, 7\}$ to a single digit: $f(6) = 1, f(5) = 2, f(3) = 3, f(1) = 4, f(7) = 5, f(4) = 6$. If $q = q_i$, δ_q will be an i digit number, specified as follows. Let $j = \min(\{j' \leq i | PC_{q_{j'}} \neq 1\} \cup \{i\})$, then the ℓth digit (from most to least significant) of δ_q is $f(PC_\ell)$ for $\ell \leq j$ and zero for $\ell > j$. For example, if $i = 4$, $PC_{q_1} = PC_{q_2} = 1$ and $PC_{q_3} = 5$, $\delta_q = 4420$. (We have omitted the possibility of $pc = 2$, since no process in $q \in Q$ has $PC_q = 2$.)

Lemma 5.4.4. *Let $q \in Q$ be any process in the wait-queue, $\delta_q = 1$ if and only if q is in the Critical Section.*

Proof. If $\delta_q = 1$, then it has only one digit and thus $q = q_1$. So, $j = 1$ and $PC_q = 6$, meaning that p is in the Critical Section.

If q is in the Critical Section, it must be the case that $q = q_1$ by I_{13}. So, $\delta_q = 1$ since $j = 1$ and $PC_{q_1} = PC_q = 6$. □

We now show that δ_q will eventually go down if q does not abort.

Lemma 5.4.5. *Let $q = q_i \in Q$ not be in the Try Section, and let $j = \min(\{j' \leq i | PC_{q_{j'}} \neq 1\} \cup \{i\})$. Let π be the next process to take a step. We define the critical set $C(q) = \{q_\ell \mid \ell \in [1, j]\}$.*

- *If π is not in the critical set, then δ_q does not increase and the critical set remains unchanged after π's step.*
- *If π is in the critical set, then δ_q strictly decreases.*

Proof. We show each of the subparts of the lemma

- If $\pi \notin C(q) = \{q_1, \ldots, q_j\}$, then it it cannot change PC_{q_ℓ} for $\ell \in [1, j]$, so the first j digits of δ_q are unchanged. Furthermore, we observe from the proof of Theorem 5.4.1, that the index i such that $q = q_i$ cannot increase. So, δ_q cannot increase and the critical set is unchanged.

- If $\pi \in C(q)$, then we consider cases. If $\pi = q_\ell \in \{q_1, \ldots, q_{j-1}\}$, then $PC_{q_\ell} = 1$ before the step by definition of δ_q and $*mynode_{q_\ell} = pred_{q_\ell}$ by I_6. So, PC_{q_ℓ} will become three and thus ℓth digit of δ_q will change from four to three. The remaining preceding digits will remain the same and the subsequent digits will become zero, so δ_q will decrease in value. The remaining case is $\pi = q_j$. If $PC_{q_j} \in \{3, 5, 6\}$, then q_j's step will make q_i become q_{i-1} since $PC_{q_{j-1}} = 1$ and thus holds $*mynode_{q_{j-1}} = pred_{q_{j-1}}$ by I_6. This will abortable-mutex:reduce the number of digits in δ_q and thereby make it strictly smaller. If $PC_{q_j} \in \{4, 7\}$, then the jth digit of δ_q—$f(PC_{q_j})$—will decrease due to the step, and all the higher order digits will remain the same. So, δ_q will strictly decrease in all cases.

□

Starvation freedom follows directly from the above two lemmas.

Theorem 5.4.6. *Algorithm 7 satisfies starvation freedom.*

Proof. Let q be a process that starts an attempt that it will not abort. It will join the process wait-queue Q in at most two steps (by the time it is in line 3) by I_9. Let m be the minimum value that δ_q attains. m cannot be greater than one, since $q_j \in C(q)$ is in the Try or Critical Section and thus will eventually take a step thereby reducing δ_q further by Lemma 5.4.5. Therefore, $m = 1$ and q does reach the Critical Section by Lemma 5.4.4. □

It is clear from the algorithm that both the Exit and Abort Section take worst case constant time to execute. We want to further show that All three sections of code, Try, Exit, and Abort take only amortized constant time. To show this, we use a potential function Φ. Since lines 4 and 5 are in a while-loop, we must ensure that they have zero amortized cost. An execution of line 5 by a process p always costs one RMR for the FAS, so we define the indicator $\mathbb{1}_{\{PC_p=5\}}$, and will include it in the potential function. If $*pred_p$ is not in the cache, an execution of line 4 costs one RMR and may also lead to an execution of line 5, otherwise it costs nothing. So, we define the function $\overline{C}_p$ which indicates whether $*pred_p$ is *not* in p's cache. Finally, we observe that if p's node is aborted, i.e., if $*mynode_p = pred_p$, then this increases the potential. So, we define $\mathcal{A}p$ to be an

indicator that p's node is aborted. We now define the potential function as:

$$\Phi \triangleq \sum_{p \in P} (2A_p + \mathbb{1}_{\{PC_p = 5\}}) + \sum_{i \in [1,k]} 2\overline{C}_{q_i}$$

Lemma 5.4.7. *Let $\alpha(\ell)$ be the amortized cost of line $\ell \in [1,7]$. Then, $\alpha(\ell)$ is bounded by a constant for lines $[1,3] \cup [6,7]$, and $\alpha(\ell) \leq 0$ for $\ell \in [4,5]$.*

Proof. We will prove the invariant by induction on steps of the multiprocess system. In particular, we consider what happens when a process π executes its next step.

line 1: The real cost of the line is one due to the FAS. If $\pi \in Q$ and it has at most one successor by the invariant, and thus the change in potential is at most 2 due to the $\overline{C}$ function. So, amortized cost is $\alpha(1) \leq 3$.

line 2: The real cost of the line is one due to the FAS, and the potential function is unchanged because the newly joining process has no successor. So, the amortized cost is $\alpha(2) = 1$.

line 3: The real cost of this line is one due to the FAS. The potential function may increase by two, since swapping into $*pred_\pi$ ensures that $\overline{C}_\pi$ is true. So, the amortized cost is $\alpha(3) \leq 3$.

line 4: There are two cases for this line. If $*pred_\pi$ is already in the cache, then this line costs nothing, and there is no change to the potential. If $*pred_\pi$ is not in the cache, then the real cost of the line is 1. We know however that $\pi = q_i$ for some $i \in [1,k]$ by I_9. So, the two units of potential that are assigned to q_i since $*pred_{q_i}$ was not in its cache can be utilized to pay for the real cost, and for (possibly) entering line 5. So, the amortized cost of this line is $\alpha(4) \leq 0$.

line 5: The real cost of this line is one due to the FAS. However, one unit of potential is released since π will no longer be on line 5; it will go to line 6 if the while condition fails, and 4 or 7 otherwise.

We now show that the rest of the potential function is unchanged by the line. In particular, if $*pred_\pi$ was previously cached, then we know that $*pred_{q_i} = pred_{q_{i-1}}$ since its value was read on line 4, and I_{10} imposes that there this is the only non-`nil` and non-`token` value of $*pred_{q_i}$ (when combined with I_2, I_3). So, it must be the case that $A_{q_{i-1}}$ was true before line 5 but not after, and the net potential change is zero Otherwise, if $*pred_\pi$ was not cached, there is no other change to the potential function.

So, the amortized cost of this line is $\alpha(5) = 0$.

line 6: The real cost of this line is one due to the write. Since π is bound to go to the Remainder at the conclusion of this line by the mutual exclusion property, there is no other change to the potential function. So the amortized cost of this line is $\alpha(6) = 1$.

line 7: The real cost of this line is one for the write operation. The change in potential is up to four, since the write makes A_π true, and possibly causes $\overline{C}_{q_{i+1}}$ to become true if $\pi = q_i \in Q$. So, the amortized cost of this line is $\alpha(7) \leq 5$.

□

We use the above Lemma to prove the main RMR efficiency theorem about the CC algorithm.

Theorem 5.3.2. *Algorithm 7 solves Abortable Mutual Exclusion for an unbounded number of processes of arbitrary names using the FAS synchronization primitive. In particular, the algorithm satisfies Mutual Exclusion, Bounded Exit ($O(1)$ steps), Fast Abort, Starvation-Freedom, and AFCFS. It uses only $O(1)$ space per process, and in the CC model has $O(1)$ amortized RMR complexity.*

Proof. All claims except for RMR complexity are immediate from the prior lemmas. To show that the amortized RMR complexity is $O(1)$, we simply observe that the amortized cost of an execution of the protocol is bounded by the sum of $\sum_{\ell \in \{1,2,3,6,7\}} \alpha(\ell)$, since the two lines (line 4 and 5) in the loop have zero amortized cost by Lemma 5.4.7. This quantity is constant, also by Lemma 5.4.7. □

5.5 An Amortized O(1) RMR Algorithm for CC and DSM

In this section, we present our abortable mutual exclusion algorithm that has $O(1)$ amortized RMR complexity for both the CC and DSM models. To help the reader understand the algorithm, below we first present the high level ideas and only later point the reader to the actual algorithm and our line-by-line commentary.

5.5.1 Intuitive Description of the Main Ideas and Their Representation

Our algorithm, once again, is essentially a queue lock: in the Try section, as processes wait to acquire the lock, they wait in a queue. When a process enters the Try section, it adds itself to

the end of the queue. The process that is at the front of this queue is the one that enters the CS. When a process leaves the CS, it removes itself from the queue, and lets the next process, which is now at the front of the queue, enter the CS. In our informal description and in the statement of our invariant, we let Q denote this abstract "process-queue", $k \geq 0$ denote the number of processes in Q, and $q_1, q_2, \ldots, q_k$ denote the sequence of processes in Q, where q_1 is the front process and q_k is the tail process.

This abstract process-queue Q is represented in the algorithm by a list of nodes. A node is simply a single word of shared memory. If a set P consisting of n processes participate in the algorithm (some of which are in the Remainder section and the others active in Try, Critical, Exit, or Abort sections), then the algorithm employs a total of $n+1$ nodes, of which n nodes are owned by the n processes and the remaining node is not owned by any process. Thus, at any point, each process owns one node, with different processes owning different nodes. However, the node that a process owns and the node that is not owned by any process change with time. In the algorithm, each process p has a local variable $mynode_p$ that holds the address of the node that p owns. We let N denote the set of addresses of the $n+1$ nodes.

Turning our attention back to the abstract process-queue Q of length k, it is represented in the algorithm by a list of $k+1$ nodes whose addresses we denote by $a_0, a_1, \ldots, a_k$, where $a_1, a_2, \ldots, a_k$ are the addresses of the nodes owned by $q_1, q_2, \ldots, q_k$, respectively, and a_0 is the address of the node that is not owned by any process. Thus, for all $i \in [1, k]$, $a_i = mynode_{q_i}$. We call the list $a_0, a_1, \ldots, a_k$ the "node-queue", which closely corresponds to but is distinct from the process-queue Q. We say q_{i+1} is q_i's *successor process* and q_{i-1} is q_i's *predecessor process*. Similarly, we call a_{i+1} is q_i's *successor node* and a_{i-1} is q_i's *predecessor node*. Process q_k has no successor node, but q_1 has a_0 as its predecessor node. Henceforth, we simply use the terms successor and predecessor, and let it be inferred from the context whether we are referring to the process or to the node.

In the algorithm, each process p has another local variable $pred_p$ through which p remembers the address of its predecessor node. In particular, for all $i \in [1, k]$, $pred_{q_i} = a_{i-1}$. There is also a shared variable TAIL, which holds a_k, the address of the last node in the node-queue. When a process p enters the Try section, it performs a FAS(TAIL, $mynode_p$) and stores the return value in $pred_p$ so as to both add itself to the end of the queue and simultaneously remember its predecessor.

The node a_0 is the only node that may contain a special value denoted `token`. Once a process p enters the Try section and adds itself to the queue, it checks if its predecessor node contains `token`. If it does, p knows it is q_1 and enters the CS. Otherwise, p busywaits until its predecessor

will inform p that p's turn to the enter the CS has come up. In the algorithm, there is a shared variable called go_p for this purpose—it is on this variable that p busywaits. For p's predecessor to later inform p that it may enter the CS, the predecessor needs to know the address of go_p, so p deposits go_p's address in its predecessor node before busywaiting.

The final high level idea concerns how a process aborts. When a process q_i in the Try section wishes to abort, it leaves its node a_i in the queue, but marks the node as "aborted" by writing the address of its predecessor node into its node, i.e., by writing a_{i-1} into $*a_i$. By doing this writing with a FAS, q_i simultaneously learns the address of $go_{q_{i+1}}$, where its successor busy-waits, and wakes up the successor. The successor q_{i+1} then reads its predecessor node a_i, where it sees the address a_{i-1} and infers that q_i has aborted, so splices out a_i from the queue by writing `nil` in $*a_i$, and henceforth regarding a_{i-1} as its predecessor.

Suppose that after q_i aborted, it decides to invoke the Try section in a bid to acquire the lock. Since it is possible that its node is not yet spliced out of the queue by its successor, q_i tries to reclaim its old spot in the queue by simply switching the value in its node from a_{i-1} to `nil`. If it notices that its node was already spliced out, the process adds its node to the end of the queue as in the normal course. Otherwise, it has happily reclaimed its old spot in queue.

5.5.2 The algorithm and line-by-line commentary

Having elaborately described the main ideas and how they are represented and implemented by the local and shared variables, we now refer the reader to the precise algorithm (Algorithm 8) and informally explain how it works by going over the code of an arbitrary process p line by line. A note about our convention on how lines are numbered: in a single step of the execution, a process performs an operation on a single shared variable, but can perform any number of local actions. Therefore, in the figure, we numbered only those lines where an operation is performed on a *shared* variable.

Being at Line 1 amounts to being in the Remainder section. At Line 1, p is unsure whether it had aborted its previous attempt, but if it had aborted, p knows that it would have left $pred_p$ in its node. So, to reclaim its old spot in the queue (in the event that it aborted its previous attempt and its node has not yet been spliced out of the queue by its successor), p performs a FAS on its node (Line 1). If the FAS returns $pred_p$, p is sure it has reclaimed its old spot in the queue and proceeds to Line 3. Otherwise, p realizes that either it didn't abort its previous attempt or its aborted node has since been spliced out, so p appends its node afresh to the queue and records its

predecessor node in $pred_p$ (Line 2). Once in the queue, p performs a FAS on its predecessor node to simultaneously inform the predecessor of the address of its busy-wait variable and learn the value v_p in the predecessor node (Line 3). If v_p is `token`, p infers that it is q_1, the front process in the wait-queue, so it terminates the Try section and proceeds to the CS. If v_p is non-`nil` and not the address of p's busy-wait variable, then it must be the case that the predecessor aborted and v_p has the address of the predecessor's predecessor. In this case, p knows that it spliced its predecessor out of the queue, and updates its predecessor (this shortens the queue by one node, which is crucial to proving starvation-freedom). Having updated its predecessor, p proceeds to Line 6 to check what is in store at this new predecessor. In the remaining case (when v_p is either `nil` or the address of its busy-wait variable), p understands it has no option but to wait until woken by its predecessor. So, it busy-waits (Line 4) and, once woken by its predecessor, resets go_p to prepare it for any busy-wait in the future (Line 5), and then moves on to inspect the predecessor node (Line 6) to determine why the predecessor woke it up.

Once p leaves the CS, it deposits the token in its node to signal its successor that it may enter the CS (Line 7). If the FAS operation at Line 7 returns a non-`nil` value, p knows that the value must be the address where the successor is busy-waiting. So, p wakes up its successor (Line 8). Importantly, the moment p deposits `token` in its node at Line 7, that node becomes the new a_0, the token holding node that does not belong to any process, and p grabs its predecessor node (the old a_0) as its own node.

Before describing the Abort section, we note that in our algorithm, after p receives the abort signal from the environment, it is allowed to jump to the Abort section only after performing Line 3, or any iteration of Line 4, or Line 5, or Line 6. In particular, if the abort signal comes from the environment when p is at Line 3, it is required to execute Line 3 and all local actions associated with Line 3 before jumping to the Abort section. Furthermore, if the abort signal comes while p is looping at Line 4, p does not have to wait until the wait-till loop terminates before jumping to the Abort section; it can jump to the Abort section immediately (i.e., regardless of whether go_p is true or false).

At the start of the Abort section, p erases the address of its busy-wait variable from its predecessor node because p is on the way out and no longer wants to be woken by the predecessor (Line 9). However, if p observes the token in the predecessor node, p knows that it has the permission enter the CS now. However, since p wishes to abort, it will sidestep CS and proceed directly to the Exit section (and complete its abort by executing the Exit section and returning to the Remainder from

there). Another possibility is that $v_p \neq \&go_p$, which means that p's predecessor aborted, in which case p updates its predecessor to v_p, which holds p's predecessor's predecessor. At this point, p marks its node as aborted by writing in it p's predecessor (Line 10). A non-nil return value would be the address where p's successor is busy-waiting, so p informs the successor of its departure by setting the successor's busy-wait variable (Line 11).

5.6 Proof of Correctness

We state the invariant of Algorithm 8 in Figure 5-2, and prove its correctness below.

Theorem 5.6.1. *The statement I, which is the conjunction of $I_1, I_2, \ldots, I_{12}$ in Figure 5-2, is an invariant of Algorithm 8. Furthermore, the quantities k, A, and Q in the invariant I are unique.*

Proof. We will prove the invariant by induction on steps of the multiprocessor system. In particular, we consider what happens when a process π executes its next step.

Base Case: At the beginning $k = 0$ and a_0 is the address of the node that initially belongs to no process. I_1, and I_4 hold true since TAIL $= a_0$. I_2, I_3, I_7, I_{10}, and I_{11} hold trivially since $k = 0$. I_6 holds trivially since $PC_p = 1$ for every $p \in P$. I_5 holds since $pred_p = mynode_p \in N$ for every $p \in P$. I_8 holds since $PC_p = 1$ and $*mynode_p = \texttt{nil} \neq mynode_p = pred_p$ for every $p \in P$. I_9 holds since $k = 0$ and $*a_0 = \texttt{token}$.

Induction Step: We assume that invariant holds in a particular configuration, and consider what happens if the next step is taken by some process π executing one of the eleven possible lines. We use primed variables to reflect the truth after the step, and unprimed variables before when there is ambiguity.

line 1: We consider two cases: $*mynode_\pi = pred_\pi$ and $*mynode_\pi \neq pred_\pi$.

If $*mynode_\pi = pred_\pi$, then $\pi \in Q$ by the contrapositive of I_8. So let $\pi = q_i$. So, the comparison on line 1 will fail and PC_π will become 3. I_7 will hold since $*mynode_{q_i} = \texttt{nil}$. The rest of the invariants will continue to hold since they are unaffected.

If $*mynode_\pi \neq pred_\pi$, then $\pi \notin Q$ by I_7. So, the comparison on line 1 will succeed and PC_π will become 2. I_6 and I_8 continue to hold since $\pi \notin Q$ and $*mynode_\pi = \texttt{nil}$. The rest of the invariants will continue to hold since they are unaffected.

line 2: By I_6 we know $\pi \notin Q$ before the line execution. After the execution, $q_1, \ldots, q_k$ and $a_0, \ldots, a_k$ will remain unchanged. The value of k' will be $k+1$, with $\pi = q_{k'}$ and $mynode_\pi = a_{k'}$. Finally, $PC'_\pi = 3$. I_1, I_2, and I_3 will continue to hold by the FAS on line 2. I_5 continues to hold since $pred_\pi = a_K \in N$ by I_4. I_7 continues to holds for $q_{k'}$ since $*mynode_{q_{k'}} = \texttt{nil}$; I_7 continues to hold for the other q_i's since their program counters are unaffected, and we notice that $\texttt{nil} \in \{\texttt{nil}, \&go_{q_{i+1}}\}$. I_9 continues to hold since if $k' = 1$, the $k = 0$ before line 2, and otherwise it is unaffected by the line. The rest of the invariants will continue to hold since they are unaffected.

line 3: By the contrapositive of I_8, we establish that $\pi \in Q$. So, let $\pi = q_i$ for $i \in [1, k]$. I_7 and I_9 imply that there are three cases:

1. Assume $i = 1$ and $*a_0 = \texttt{token}$. In this case, $*a_0$ becomes $\&go_{q_1}$, and $PC'_\pi = PC'_{q_1} = 7$ after π notices that the while-loop condition after line 3 (which is a local action) fails. This immediately implies I_9 continues to hold. I_{10} continues to hold since q_1 was unchanged by the execution of line 3. The rest of the invariants continue to hold since they are unaffected.
2. Assume $i > 1$ and $*a_{i-1} \in \{\texttt{nil}, \&go_{q_i}\}$. This means that line 3, will simply replace the initial contents of $*a_{i-1}$ with $\&go_{q_i}$, and $PC'_\pi = 4$. I_{11} continues to hold since $(*a_{i-1} = \&go_{q_i})$. All invariants continue to hold since they are unaffected.
3. Assume $i > 1$ and $*a_{i-1} = pred_{q_{i-1}}$. This means that line 3, will result in replacing the contents of $*a_{i-1}$ with $\&go_{q_i}$. This will also trigger the if-condition inside the while loop and $pred_\pi$ will become $pred_{q_{i-1}} = a_{i-2}$. PC'_π becomes 6. This action removes the old q_{i-1} from Q by the uniqueness established through I_1, I_2, I_3, I_4, making $k' = k-1$, and we rename $q_i, \ldots, q_k$ to $q'_{i-1}, \ldots, q'_{k-1}$. I_1, I_2 and I_3 continue to hold by the renaming. By I_6 and I_7 we establish that the old q_{i-1} had $PC_{q_{i-1}} \in \{1, 11\}$; so we notice that I_8 continues to hold for q_{i-1}. The rest of the invariants continue to hold since they are unaffected.

line 4: If $go_\pi = false$, then PC'_π remains at 4. If $go_\pi = true$, then PC'_π becomes 5. In both cases, all invariants continue to hold as they are unaffected.

line 5: go'_π becomes *false* regardless of its initial value and PC'_π becomes 6. All invariants continue to hold as they are unaffected.

line 6: The analysis of this line is precisely identical to that of line 3.

line 7: By I_{10} we establish that $\pi = q_1$. Since, the local instruction $mynode_\pi \leftarrow pred_\pi$ completes atomically along with the shared instruction, by $\pi \notin Q'$ by the uniqueness of Q' established by I_1, I_2, I_3, I_4. The old $mynode_\pi = mynode_{q_1}$ becomes the new a'_0, $k' = k - 1$ and the old $q_2, \ldots, q_k$ and renamed to $q'_1, \ldots, q'_{k-1}$, and I_1, I_2 and I_3 continue to hold by the renaming. I_4 continues to hold since $a_0 = mynode'_\pi$ and $mynode_\pi = a'_0$; the names were simply permuted. I_9 continues to hold since $*a_0 =$ `token`. By I_9, $*mynode'_\pi = *a_0 \in \{\texttt{nil}, go_\pi\}$, and so I_8 continues to hold true.

I_7 shows that $*mynode_\pi$ could have either been `nil` or $\&go_{q_2}$. We consider three cases.

1. Assume $k = 1$. In this case, I_7 establishes that v'_π would surely have become `nil` and thus π would go to the Remainder Section ($PC'_\pi = 1$). The rest of the invariants continue to hold since they are unaffected or become trivial since $k' = 0$.
2. Assume $k > 1$ and $*mynode_\pi =$ `nil`. π goes to the Remainder Section ($PC'_\pi = 1$), since $v'_\pi =$ `nil`. By I_{11} on $i = 2$, we establish that $PC_{q'_1} \neq 4$ currently; so, I_{11} continues to hold. The rest of the invariants continue to hold since they are unaffected.
3. Assume $k > 1$ and $*mynode_\pi = \&go_{q_2} = \&go_{q'_1}$. This is the only case in which $v'_\pi = \&go_{q'_1} \neq$ `nil`. This in turn implies that I_{11} will continue to hold since $PC'_\pi = 8$. The rest of the invariants continue to hold since they are unaffected.

line 8: I_{11} is self inducting in this case, since if it were true that ($PC_{q_1} = 4 \wedge go_{q_1} = \mathit{false}$) and π were indeed the process $p \in P$ that had ($PC_p = 8 \wedge v_p = \&go_{q_1}$); then, by executing line 8, π would ensure that $go_{q_1} = \mathit{true} \neq \mathit{false}$. The rest of the invariants continue to hold since they are unaffected.

line 9: By the contrapositive of I_8, we establish that $\pi \in Q$. So, let $\pi = q_i$ for $i \in [1, k]$. I_7 and I_9 imply that there are three cases:

1. Assume $i = 1$ and $*a_0 =$ `token`. In this case, $*a_0$ becomes `nil`, and $PC'_\pi = PC'_{q_1} = 7$ after π notices that the while-loop condition after line 3 (which is a local action) fails. This immediately implies I_9 continues to hold. I_{10} continues to hold since q_1 was unchanged by the execution of line 3. The rest of the invariants continue to hold since they are unaffected.
2. Assume $i > 1$ and $*a_{i-1} \in \{\texttt{nil}, \&go_{q_i}\}$. This means that line 3, will simply replace the initial contents of $*a_{i-1}$ with `nil`, and $PC'_\pi = 10$. All invariants continue to

hold since they are unaffected.

3. Assume $i > 1$ and $*a_{i-1} = pred_{q_{i-1}}$. This means that line 3, will result in replacing the contents of $*a_{i-1}$ with `nil`. This will also trigger the else-if-condition and $pred_\pi$ will become $pred_{q_{i-1}} = a_{i-2}$. PC'_π becomes 10. This action removes the old q_{i-1} from Q by the uniqueness established through I_1, I_2, I_3, I_4, making $k' = k-1$, and we rename $q_i, \ldots, q_k$ to $q'_{i-1}, \ldots, q'_{k-1}$. I_1, I_2 and I_3 continue to hold by the renaming. By I_6 and I_7 we establish that the old q_{i-1} had $PC_{q_{i-1}} \in \{1, 11\}$; so we notice that I_8 continues to hold for q_{i-1}. The rest of the invariants continue to hold since they are unaffected.

line 10: The contrapositive of I_8 implies $\pi \in Q$. Let $\pi = q_i$ for $i \in [1, k]$. I_7 will continue to hold since $PC'_\pi \in \{1, 11\}$, $\pi \in Q$, and $*mynode_\pi = pred_\pi$. I_7 shows that $*mynode_\pi$ could have either been `nil` or $\&go_{q_{i+1}}$. We consider three cases.

1. Assume $k = 1$. In this case, I_7 establishes that v'_π would surely have become `nil` and thus π would go to the Remainder Section ($PC'_\pi = 1$). The rest of the invariants continue to hold since they are unaffected.
2. Assume $k > 1$ and $*mynode_\pi =$ `nil`. In this case, v'_π would surely have become `nil` and thus π would go to the Remainder Section ($PC'_\pi = 1$). The rest of the invariants continue to hold since they are unaffected.
3. Assume $k > 1$ and $*mynode_\pi = \&go_{q_{i+1}}$. This is the only case in which $v'_\pi = \&go_{q_{i+1}} \neq$ `nil`. This in turn implies that I_{11} will continue to hold since $(*a_i = pred_{q_i}) \wedge (PC'_{q_i} = 11) \wedge (v_{q_i} = \&go_{q_{i-1}})$. The rest of the invariants continue to hold since they are unaffected.

line 11: I_{11} is self inducting in this case, since if it were true that $(PC_{q_i} = 4 \wedge go_{q_i} = false)$ and π were indeed the process q_{i+1} that had $(*a_i = pred_{q_i}) \wedge (PC'_{q_i} = 11) \wedge (v_{q_i} = \&go_{q_{i-1}})$; then, by executing line 11, π would ensure that $go_{q_i} = true \neq false$. The rest of the invariants continue to hold since they are unaffected.

□

Corollary 5.6.2. *Algorithm 8 satisfies mutual exclusion.*

Proof. I_{10} states that only process q_1 can be in the critical section. □

Theorem 5.6.3. *Algorithm 8 satisfies AFCFS where the doorway constitutes Lines 1 and 2 of the Try Section.*

Proof. In order to show this property, we prove the following stronger statement by induction: *if process $p = q_i$ in the queue and process $p' = q_j$ in the queue with $j > i$, then p started its passage before p' finished its doorway.* This statement is true initially since Q is empty. The statement continues to hold inductively whenever any process leaves Q from any position. Thus, we are left with the case where a new process p enters Q. Since this case occurs only if p executes Line 2, and thereby just finishes the doorway, it is clear that every other process in Q has started its passage before p finished its doorway in its current passage. Now in conjunction with our inductive statement, we observe that by I_{10}, only the first process in Q, q_1 , can be in the CS; that finishes the proof. □

5.6.1 Proof of starvation freedom

To prove starvation freedom, we define a distance function δ that maps each process p in the Try section to a positive integer $\delta(p)$ that represents how far away p is from entering the CS. By our definition of δ, the minimum value possible for $\delta(p)$ is 1, and it is attained exactly when p is in the CS. To show that p will eventually enter the CS if it does not abort, we prove that if $\delta(p) > 1$, there is a nonempty set $\Psi(p)$ of "promoter" processes in the Try, Exit, or Abort sections such that (i) if a process from $\Psi(p)$ takes the next step, $\delta(p)$ decreases, and (ii) if a process not in $\Psi(p)$ takes the next step, $\delta(p)$ does not increase and the promoters set $\Psi(p)$ remains unchanged. Since a process from $\Psi(p)$ must eventually take a step, $\delta(p)$ is guaranteed to eventually decrease. By repeatedly applying this argument, we see that $\delta(p)$ eventually attains the minimum value of 1, at which point p enters the CS.

Our distance function δ is based on a carefully crafted auxiliary function f that maps each process $r \in Q$ to a decimal digit, based on r's program counter PC_r and the value of its go_r variable, as follows.

There is an integer $k \geq 0$, and a sequence $A = a_0, \ldots, a_k$ of $k+1$ addresses of distinct nodes and a sequence $Q = q_1, \ldots, q_k$ of k distinct processes such that

$I_1)$ TAIL $= a_k$

$I_2)$ $\forall i \in [1,k]$, $mynode_{q_i} = a_i$

$I_3)$ $\forall i \in [1,k]$, $pred_{q_i} = a_{i-1}$

$I_4)$ $N = \{a_0\} \cup \{mynode_p \mid p \in P\}$

$I_5)$ $\forall p \in P,\ pred_p \in N$

$I_6)$ $\forall p \in P,\ PC_p \in \{2,8\} \implies p \notin Q$
$\forall p \in P,\ PC_p = 2 \implies *mynode_p = \texttt{nil}$
$\forall p \in P,\ PC_p = 8 \implies *mynode_p \in \{\texttt{nil}, \&go_p\}$

$I_7)$ If $k \geq 1$, then:
$PC_{q_k} \in \{3,4,5,6,7,9,10\} \implies *mynode_{q_k} = \texttt{nil}$
$PC_{q_k} \in \{1,11\} \implies *mynode_{q_k} = pred_{q_k}$
If $k \geq 2$, then for all $q_i \in \{q_1, \ldots, q_{k-1}\}$:
$PC_{q_i} \in \{3,4,5,6,7,9,10\} \implies *mynode_p \in \{\texttt{nil}, \&go_{q_{i+1}}\}$
$PC_{q_i} \in \{1,11\} \implies *mynode_{q_i} = pred_{q_i}$

$I_8)$ $\forall p \in P,\ p \notin Q \implies (PC_p \in \{1,2,8,11\} \wedge *mynode_p \neq pred_p)$

$I_9)$ If $(k = 0 \vee PC_{q_1} \neq 7)$ then $*a_0 = \texttt{token}$, else $*a_0 \in \{\texttt{nil}, go_{q_1}\}$

$I_{10})$ $\forall p \in P,\ p \neq q_1 \implies PC_p \neq 7$

$I_{11})$ $(PC_{q_1} = 4 \wedge go_{q_1} = false) \implies \exists p \in P,\ (PC_p = 8 \wedge v_p = \&go_{q_1})$
$\forall i \in [2,k],\ ((PC_{q_i} = 4 \wedge go_{q_i} = false) \implies$
$((*a_{i-1} = \&go_{q_i}) \vee ((*a_{i-1} = pred_{q_{i-1}}) \wedge (PC_{q_{i-1}} = 11) \wedge (v_{q_{i-1}} = \&go_{q_i}))))$
$\forall p \in P, PC_p \in \{8,11\} \implies v_p \in \{\&go_p \mid p \in P\}$

$I_{12})$ $\forall p \in P,\ PC_p = 5 \implies go_p = true$

Note: by I_4, the queue of node addresses starts with the unique node address a_0 that is no process p's $mynode_p$ and by I_1 ends with TAIL. This together with I_2 and I_3, implies that k, A, and Q are uniquely defined.

Figure 5-2: Invariant of Algorithm 8.

$$f(r) = \begin{cases} 3 & if PC_r = 1 \\ 2 & if PC_r = 3 \\ 8 & if PC_r = 4 \wedge go_r = true \\ 9 & if PC_r = 4 \wedge go_r = false \\ 7 & if PC_r = 5 \\ 2 & if PC_r = 6 \end{cases}$$

Since a process in Q cannot be at Lines 2 or 8 (by I_6), we didn't specify $f(r)$ for $PC_r \in \{2, 8\}$. For a process p in the Try section, we are now ready to define p's distance from CS $\delta(p)$, and p's promoters set $\Psi(p)$.

Definition 5.6.4 (Distance function δ and Promoters set Ψ)**.** Let p be a process in the Try section or CS (i.e., $PC_p \in \{3, 4, 5, 6, 7\}$). It follows from I_8 that $p \in Q$. Let $i \geq 1$ be p's position in Q (i.e., $p = q_i$), and let $m = \min\{j \mid 1 \leq j \leq i,\ PC_{q_j} \neq 1\}$. (Since $PC_{q_i} \in \{3, 4, 5, 6, 7\}$, m is well defined.) Then:

- p's distance from CS, $\delta(p)$, is defined as the i-digit decimal number $d_1 d_2 \ldots d_i$, where each digit d_j, for $1 \leq j \leq i$, is specified as follows:

$$d_j = \begin{cases} f(q_j) = 3 & if \mathrm{j} < \mathrm{m} \\ f(q_m) & if \mathrm{j} = \mathrm{m} \\ 0 & if \mathrm{j} > \mathrm{m} \end{cases}$$

  For example, if $i = 4$, $m = 3$, and $PC_{q_3} = 5$, then $\delta_p = 3370$.

- p's set of promoters, $\Psi(p)$, is defined by

$$\Psi(p) = \begin{cases} \{r \in P \mid PC_r \in \{8, 11\} \wedge v_r = \&go_{q_m}\} & if PC_{q_m} = 4 \wedge go_{q_m} = false \\ \{q_m\} & otherwise \end{cases}$$

The next lemma follows easily from the definition of $\delta(p)$.

Lemma 5.6.5. *For any process $p \in Q$ in the Try section or CS, $\delta(p) \geq 1$, and $\delta(p) = 1$ if and only if p is in the CS.*

The values of $\delta(p)$, $\Psi(p)$, and PC_p can change from one configuration to the next. So, in contexts such as the next lemma where we need to refer to these values in more than one configuration, we add a configuration C as an extra parameter and denote these values as $\delta(p, C)$, $\Psi(p, C)$, and $PC_p(C)$.

Lemma 5.6.6. *Suppose that a process $p \in Q$ is in the Try section in a configuration C (i.e., $PC_p(C) \in \{3, 4, 5, 6\}$) and some process π (possibly the same as p) takes a step from C. Let C' denote the configuration immediately immediately after π's step.*

1. *If* $\pi \in \Psi(p, C)$*, then* $\delta(p, C') < \delta(p, C)$.

2. *If* $\pi \notin \Psi(p, C)$*, then either* $\delta(p, C') < \delta(p, C)$ *or* $(\delta(p, C') = \delta(p, C) \wedge \Psi(p, C') = \Psi(p, C))$.

Proof. Since $PC_p(C) \in \{3, 4, 5, 6\}$, it follows from I_8 that $p \in Q(C)$. Let $p = q_i(C)$, and $\delta(p, C)$ be the i-digit decimal number $d_1 d_2 \ldots d_i$, where the digits d_j are as defined above. Let $m = \min\{j \mid 1 \leq j \leq i,\ PC_{q_j}(C) \neq 1\}$.

We prove Part (1) of the lemma in two cases:

- Suppose that $PC_{q_m}(C) = 4 \wedge go_{q_m}(C) = \mathit{false}$. Then, since $\pi \in \Psi(C)$, it follows that $PC_\pi(C) \in \{8, 11\} \wedge v_\pi(C) = \&go_{q_m}$. Therefore, π's step writes *true* in go_{q_m}, making $f(q_m, C') = 8$ (note that $f(q_m, C)$ was 9). Thus, the mth digit is less in $\delta(p, C')$ than in $\delta(p, C)$, while the other digits of $\delta(p)$ remain unchanged from C to C'. Therefore, $\delta(p, C') < \delta(p, C)$.

- Suppose that it is not the case that $PC_{q_m}(C) = 4 \wedge go_{q_m}(C) = \mathit{false}$. Since $\pi \in \Psi(C)$, it follows that $\pi = q_m$.

  Suppose that $m = 1$ and $PC_{q_m}(C) \in \{3, 6\}$. Then, it follows from I_9 that q_1's step causes it jump to Line 7. So, the first digit of $\delta(p)$ changes from $f(q_1)$ in C (which is 2) to $f(q_1)$ in C' (which is 1); therefore, $\delta(p, C') < \delta(p, C)$.

  Suppose that $m > 1$ and $PC_{q_m}(C) \in \{3, 6\}$. Since $PC_{q_{m-1}} = 1$, it follows from the second part of I_7 that $^*mynode_{q_{m-1}} = pred_{q_{m-1}}$. Therefore, q_m's step shortens the queue by one, causing p's position in Q to change from i in C to $i - 1$ in C'. Thus, $\delta(p, C')$ has one fewer digit than $\delta(p, C)$. Therefore, $\delta(p, C') < \delta(p, C)$.

  Suppose that $PC_{q_m}(C) = 7$. Then, m must be 1 (by I_{10}) and q_1's step causes $q_1(C)$ to be no longer in Q in C', thereby causing p's position in Q to change from i in C to $i - 1$ in C'. Thus, $\delta(p, C')$ has one fewer digit than $\delta(p, C)$. Therefore, $\delta(p, C') < \delta(p, C)$.

  If $PC_{q_m}(C)$ is anything else (i.e., $PC_{q_m}(C) \in \{4, 5, 7, 9, 10, 11\}$), the function f is so defined that because of the changed value of q_m's program counter, $f(q_m)$ is less in C' than in C. Thus, the mth digit of $\delta(p, C')$ is less than the mth digit of $\delta(p, C)$ (while all other digits of $\delta(p, C')$ are respectively the same as those of $\delta(p, C)$). Therefore, $\delta(p, C') < \delta(p, C)$.

For the proof of Part (2) of the lemma, we consider the same two cases.

- Suppose that $PC_{q_m}(C) = 4 \wedge go_{q_m}(C) = \mathit{false}$, and $\pi \notin \Psi(C)$.

If $\pi = q_m$, q_m's step will not change the configuration (i.e., $C' = C$); therefore, $\delta(p, C') = \delta(p, C)$ and $\Psi(p, C') = \Psi(p, C)$.

If $\pi = q_j$ for some $j < m$, then q_j's PC changes from 1 to 3. So, by the definition of f, the jth digit of $\delta(p)$ changes from 3 in C to 2 in C', while all more significant digits of $\delta(p, C')$ are respectively the same as those of $\delta(p, C)$. Therefore, $\delta(p, C') < \delta(p, C)$.

If π is different from all of $q_1, q_2, \ldots, q_m$, then $\delta(p, C') = \delta(p, C)$ and $\Psi(p, C') = \Psi(p, C)$.

- Suppose that it is not the case that $PC_{q_m}(C) = 4 \wedge go_{q_m}(C) = \mathit{false}$, and $\pi \notin \Psi(C)$. Then $\pi \neq q_m$. If $\pi = q_j$ for some $j < m$, then q_j's PC changes from 1 to 3 and, as just argued, $\delta(p, C') < \delta(p, C)$. If π is different from all of $q_1, q_2, \ldots, q_m$, then $\delta(p, C') = \delta(p, C)$ and $\Psi(p, C') = \Psi(p, C)$.

□

The previous lemma helps us prove that the algorithm is starvation free.

Theorem 5.6.7 (Starvation Freedom)**.** *If a process $p \in Q$ is in the Try section (i.e., $PC_p \in \{3, 4, 5, 6\}$) and does not abort, it eventually enters the CS.*

Proof. Let C be a configuration where $PC_p \in \{3, 4, 5, 6\}$. Since p is not in the CS in C, , it follows from Lemma 5.6.5 that $\delta(p) > 1$ in C. By Lemma 5.6.6, as processes take steps from C, $\delta(p)$ never increases. It cannot remain the same forever because some process in $\Psi(p, C)$ eventually takes a step, causing $\delta(p)$ to decrease (by Lemma 5.6.6). Thus, $\delta(p)$ keeps decreasing as the execution progresses, until eventually hitting the minimum possible value of 1, at which point p is in the CS (by Lemma 5.6.5). □

5.7 Complexity Analysis

In this section we analyze the space and RMR complexities of Algorithm 8.

Theorem 5.7.1. *Algorithm 8 uses $O(1)$ space plus $O(1)$ additional space per process that participates in the algorithm.*

Proof. Clear from inspection of the "Variables" description at the start of the algorithm. □

We now wish to show that a process p performs only an amortized constant number of RMRs per attempt in Algorithm 8 in both the CC and DSM cost models. (An attempt by a process lasts

from when it leaves the Remainder to when it re-enters the Remainder.) We analyze complexity by the potential method, by defining two different potential functions Φ_{CC} and Φ_{DSM}.

We start by motivating the definition of Φ_{DSM}, the simpler of the functions. Since we are proving a constant bound, we must show that each iteration of p's while-loop (lines 4, 5, and 6) is paid for by some other action. Since, go_p is in p's partition of memory, lines 4 and 5 have no cost in the DSM model, so we focus on line 6. The two ways that p can get to line 6 are: (1) go_p becomes *true* when p is busy-waiting on line 4, and (2) a FAS on either line 3 or 6 successfully removes an aborted node from the linked list. So, we charge the executions of line 6 to processes that write *true* to go_p or abort a node by writing $pred_p$ in $*mynode_p$. This gives rise to the definition:

$$\Phi_{DSM} = \sum_{p \in P} \mathbb{1}_{\{PC_{go_p=true}=5\}} + \mathbb{1}_{\{PC_{PC_p=6}=5\}} + \mathbb{1}_{\{PC_{*mynode_p=pred_p}=5\}}$$

In this definition $\mathbb{1}_{\{PC_{prop}=5\}}$ is an indicator—it equals one if *prop* is *true* and zero otherwise. Note that Φ_{DSM} is a *proper potential function*, since it is zero in the initial configuration and always non-negative. We now state and prove a lemma that bounds the amortized cost in the DSM model of line ℓ, $\alpha_{DSM}(\ell)$, for each $\ell \in [1, 11]$.

Lemma 5.7.2. *$\alpha_{DSM}(\ell) \leq 1$ for lines $\ell \in [1, 3] \cup \{7, 9\}$, $\alpha_{DSM}(\ell) \leq 2$ for lines $\ell \in \{8, 10, 11\}$, and $\alpha_{DSM}(\ell) \leq 0$ for $\ell \in [4, 6]$.*

Proof. We will prove the invariant by induction on steps of the multiprocessor system. In particular, we consider what happens when a process π executes its next step.

line 1: The real cost of the line is zero or one depending on whether $*mynode_p$ is in π's partition. This line can only decrease the potential (by decreasing the indicator $\mathbb{1}_{\{PC_{*mynode_p=pred_p}=5\}}$). So, amortized cost is $\alpha_{DSM}(1) \leq 1$.

line 2: The real cost of the line is one due to the FAS, and the potential function is unchanged. So, $\alpha_{DSM}(2) = 1$.

lines 3,7, and 9: The real cost of the line is zero or one depending on whether the FAS is on a node in π's partition. This line can only decrease the potential (by decreasing the indicator $\mathbb{1}_{\{PC_{*mynode_p=pred_p}=5\}}$). So, $\alpha_{DSM}(3)$, $\alpha_{DSM}(7)$, $\alpha_{DSM}(9) \leq 1$.

line 4: The real cost of this line is zero since go_p is in p's partition. The potential change is also zero. So, $\alpha_{DSM}(4) = 0$.

line 5: The real cost of this line is zero since go_p is in p's partition. When this line is executed, $\mathbb{1}_{\{PC_{go_\pi = true}=5\}}$ indicator must decrease by one due to I_{12} and $\mathbb{1}_{\{PC_{PC_\pi=6}=5\}}$ must increase by one. So, the potential change is zero. So, $\alpha_{DSM}(5) = 0$.

line 6: The real cost of the line is zero or one depending on whether $*pred_p$ is in π's partition. Here we have two cases. If $*pred_\pi = mynode_p$ for some process p and $*mynode_p = pred_p$, then we use the indicator $\mathbb{1}_{\{PC_{*mynode_p=pred_p}=5\}}$ to pay for the real cost (since PC_π will become 6 again). Otherwise, PC_π will end up at some other line, and we pay for the real cost using the potential drop caused by the indicator $\mathbb{1}_{\{PC_{PC_\pi=6}=5\}}$. So, in either case the amortized cost of this line is $\alpha_{DSM}(6) \leq 0$.

line 8, and 11: The real cost of the line is zero or one depending on whether $*v_\pi$ is in π's partition. The potential change is at most one, since at most one indicator $\mathbb{1}_{\{PC_{go_\pi = true}=5\}}$ can go high. So, $\alpha_{DSM}(8), \alpha_{DSM}(11) \leq 2$.

line 10: The real cost of the line is zero or one depending on whether the FAS is on a node in π's partition. The potential can go up by at most one since $*mynode_\pi = pred_\pi$ after the line. So, $\alpha_{DSM}(10) \leq 2$.

□

The CC model is more complicated to analyze than the DSM model since: (1) line 4 can now have a real cost if go_p is not in p's cache, and (2) line 5 always has a real cost, since it is a write operation, and causes go_p to become uncached. To deal with (1), we define $\overline{C}_p$ to be the predicate that go_p is not in p's cache, and add the indicator $\mathbb{1}_{\{PC_{\overline{C}_p}=5\}}$ to Φ_{CC}. To deal with (2), we simply multiply the weight of the indicator that $go_p = true$, to pay for the additional costs incurred on line 5. This results in the definition:

$$\Phi_{CC} = \sum_{p \in P} \big(3 \times \mathbb{1}_{\{PC_{go_p=true}=5\}} + \mathbb{1}_{\{PC_{PC_p=6}=5\}} + \mathbb{1}_{\{PC_{*mynode_p=pred_p}=5\}} + \mathbb{1}_{\{PC_{\overline{C}_p}=5\}}\big)$$

At the cost of charging one unit to a process that is newly joining the protocol, we think of go_p as initially residing in p's cache; so, Φ_{CC} is also a proper potential function. We define $\alpha_{CC}(\ell)$ as the amortized cost in the CC model of line ℓ of Algorithm 8, and state and prove a lemma below (analogous to Lemma 5.7.2) for the CC model.

Lemma 5.7.3. *$\alpha_{CC}(\ell)$ is bounded by a constant for all $\ell \in [1,11] - [4,6]$, and $\alpha_{CC}(\ell) \leq 0$ for $\ell \in [4,6]$.*

Proof. We will prove the invariant by induction on steps of the multiprocessor system. In particular, we consider what happens when a process π executes its next step.

line 1: The real cost of the line is one due to the FAS. The change in potential is non-positive. So, amortized cost is $\alpha(1) \leq 1$.

line 2: The real cost of the line is one due to the FAS, The change in the potential is zero. So, the amortized cost is $\alpha(2) = 1$.

lines 3, 7, and 9: The real cost of this line is one due to the FAS. The change in potential is once again non-positive. So, the amortized cost is $\alpha_{CC}(3), \alpha_{CC}(7), \alpha_{CC}(11) \leq 1$.

line 4: There are two cases for this line: either go_π is in π's cache, or not. If go_π is cached, then both the real and amortized costs are zero. If go_π is not cached, then the real cost is one, but is cancelled out by the drop in potential caused by the fact that go_π becomes cached. So, the amortized cost of this line is $\alpha_{CC}(4) \leq 0$.

line 5: The real cost of this line is one due to the writing of *false*. Additionally, go_π becomes uncached (if it was previously cached), thereby causing a one unit potential increase; and PC_π becomes 6, causing the corresponding indicator to become one. However, by I_{12}, there is a three unit potential drop due to go_π becoming *false*. So, the amortized cost of this line is $\alpha_{CC}(5) = 0$.

line 6: The real cost of the line is one due to the FAS. Here we have two cases. If $*pred_\pi = mynode_p$ for some process p, and $*mynode_p = pred_p$, then we use the indicator $\mathbb{1}_{\{PC_{*mynode_p = pred_p} = 5\}}$ to pay for the real cost (since PC_π will become 6 again). Otherwise, PC_π will end up at some other line, and we pay for the real cost using the potential drop caused by the indicator $\mathbb{1}_{\{PC_{PC_\pi = 6} = 5\}}$. So, in either case the amortized cost of this line is $\alpha_{CC}(6) \leq 0$.

lines 8 and 11: The real cost of this line is one for the write operation. Since $*v_\pi$ is a *go*-variable (by I_{11}) being set to *true*, this line can cause a potential increase of three units. So, the amortized cost of this line is $\alpha_{CC}(8), \alpha_{CC}(11) \leq 4$.

line 10: The real cost of the line is one due to the FAS. Since $*mynode_\pi$ becomes $pred_\pi$ due to this line, there is a possible potential increase of one unit. So, the amortized cost of this line is $\alpha_{CC}(10) \leq 2$.

□

The following theorem summarizes the properties of Algorithm 8.

Theorem 5.7.4 (Main Theorem)**.** *Algorithm 8 solves Abortable Mutual Exclusion for an unbounded number of processes of arbitrary names using the FAS synchronization primitive. In particular, the algorithm satisfies Mutual Exclusion, Bounded Exit ($O(1)$ steps), Fast Abort, Starvation-Freedom, and AFCFS. It uses only $O(1)$ space per process and, in both the CC and the DSM models, has $O(1)$ amortized RMR complexity.*

Proof. All claims except for RMR complexity and Fast Abort are immediate from the prior lemmas.

In both the CC and DSM models, all the lines (4, 5, and 6) that appear in the while-loop of Algorithm 8 have zero amortized cost, and the remaining lines have constant amortized cost in the DSM model (by Lemma 5.7.2), and in the CC model (by Lemma 5.7.3). Since lines that are not in the loop are executed at most once per attempt, the amortized cost of an attempt is $O(1)$ in both the CC and DSM models.

Regardless of when the environment sends an abort signal to process p, process p can reach either the Exit or Abort section within three shared memory instructions, and either of these sections takes at most three more shared memory instructions. So, aborting happens within six shared instructions, thereby the algorithm satisfies Fast Abort. □

5.8 Model Checking

In addition to our rigorous mathematical proofs, we have also modeled checked Algorithm 7. That is, we modeled the algorithm in the Temporal Language of Actions (TLA), and used the TLA+ toolbox's model checker, TLC, to confirm via brute-force search that the algorithm indeed satisfies: mutual exclusion, AFCFS, Starvation Freedom, and type correctness. We model checked Mutual exclusion and type correctness for systems of up to five processes, and AFCFS and Starvation Freedom for systems of up to three processes. The first test resulted in searching through a state graph with over 20 million distinct states, and the second resulted in analyzing a graph of over 85

thousand distinct states. The original TLA file is publicly accessible at: https://github.com/visveswara/machine-certified-linearizability/blob/master/AbortableQueueLockCC.tla

5.9 Concluding Remarks

In this chapter, we have shown how the well known ideas behind a queue lock can be enhanced to efficiently implement an abortable lock for both the CC and DSM models. However, the constant RMR complexity that we achieved applies only in the amortized case and not in the worst-case. A natural open problem is to find or refute the existence of a deterministic worst-case constant RMR abortable lock, or of even a randomized constant expected RMR abortable lock, using common synchronization primitives such as FAS, CAS, and Fetch-and-Add.

Algorithm 8 : Amortized constant RMR abortable lock for CC and DSM machines. Code shown for process p. p jumps to the Abort Section if the abort signal is on and p is at Line 4 or 5 or 6.

Variables

- A *node* is a single shared memory word that can hold a pointer, or `nil`, or `token`.
- TAIL: shared variable that points to a node. Initially, it points to a node that is allocated and initialized to `token`.
- When a process p first participates in the algorithm:
  - It allocates two local variables $mynode_p$ and $pred_p$ both of which point to the same freshly allocated node initialized to `nil`.
  - It allocates a shared boolean go_p initialized to *false*; on a DSM machine, this variable must be in p's partition of shared memory.
  - It allocates an uninitialized local variable v_p that can hold a pointer, `nil`, or `token`.

Section TRY(p)

1: **if** FAS($*mynode_p$, `nil`) $\neq pred_p$ **then**
2: $pred_p \leftarrow$ FAS(TAIL, $mynode_p$)
3: $v_p \leftarrow$ FAS($*pred_p$, $\&go_p$)
while $v_p \neq$ `token` **do**
if $v_p \notin \{$`nil`, $\&go_p\}$ **then**
$pred_p \leftarrow v_p$
else
4: **wait till** $go_p = true$
5: $go_p \leftarrow false$
6: $v_p \leftarrow$ FAS($*pred_p$, $\&go_p$)

Section EXIT(p)

7: $v_p \leftarrow$ FAS($*mynode_p$, `token`)
$mynode_p \leftarrow pred_p$
if $v_p \neq$ `nil` **then**
8: $*v_p \leftarrow true$

Section ABORT(p)

9: $v_p \leftarrow$ FAS($*pred_p$, `nil`)
if $v_p =$ `token` **then**
goto EXIT (line 7)
else
if $v_p \notin \{$`nil`, $\&go_p\}$ **then**
$pred_p \leftarrow v_p$
10: **if** $(v_p \leftarrow$ FAS($*mynode_p$, $pred_p$)) $\neq$ `nil` **then**
11: $*v_p \leftarrow true$

Remark 5.5.1. *As with the previous algorithm, we can use the address* &TAIL *as* `token`.

Chapter 6

Recoverable Mutual Exclusion

6.1 Introduction

In light of recent advances in non-volatile main memory technology, Golab and Ramaraju reformulated the traditional mutex problem into the novel *Recoverable Mutual Exclusion* (RME) problem. The best known algorithm for RME, due to Golab and Hendler [76], has sub-logarithmic remote memory reference (RMR) complexity for Cache-Coherent (CC) multiprocessors, but unbounded RMR complexity for Distributed Shared Memory (DSM) multiprocessors. In this chapter, we present an algorithm that ensures the same sublogarithmic bound as theirs for both CC and DSM multiprocessors, besides possessing some additional desirable properties. In the rest of this section, we describe the model, the RME problem, the complexity measure, and then describe this chapter's contribution in the context of prior work.

6.1.1 The Model

The advent of *Non-Volatile Random Access Memory* (NVRAM) [172][188][200] — memory whose contents remain intact despite process crashes — has led to a new and natural model of a multiprocessor and spurred research on the design of algorithms for this model. In this model, asynchronous processes communicate by applying operations on shared variables stored in an NVRAM. A process may crash from time to time. When a process π crashes, all of π's registers lose their contents: specifically, π's program counter is reset to point to a default location ℓ in π's program, and all other registers of π are reset to $\perp$; however, the shared variables stored in the NVRAM are unaffected by a crash and retain their values. A crashed process π eventually restarts, executing the program

beginning from the instruction at the default location ℓ, regardless of where in the program π might have previously crashed.

When designing algorithms for this model, informally the goal is to ensure that when a crashed process restarts, it reconstructs the lost state by consulting the shared variables in NVRAM. To appreciate that this goal can be challenging, suppose that a process π crashes when it is just about to perform an operation such as $r \leftarrow \text{FAS}(X, 5)$, which fetches the value of the shared variable X into π's register r and then stores 5 in X. If a different process π' performs $\text{FAS}(X, 10)$ and then π restarts, π cannot distinguish whether it crashed immediately before or immediately after executing its FAS instruction.

6.1.2 The Recoverable Mutual Exclusion (RME) problem

In the Recoverable Mutual Exclusion (RME) problem, there are n asynchronous processes, where each process repeatedly cycles through four sections of code—Remainder, Try, Critical, and Exit sections. An *algorithm* (for RME) specifies the code for the Try and Exit sections of each process. Any process can execute a *normal step* or a *crash step* at any time. In a normal step of a process π, π executes the instruction pointed by its program counter PC_π. We assume that if π executes a normal step when in Remainder, π moves to Try; and if π executes a normal step when in CS, π moves to Exit. A crash step models the crash of a process and can occur regardless of which section of code the process is in. A crash step of π sets π's program counter to point to its Remainder section and sets all other registers of π to $\perp$.

A *run* of an algorithm is an infinite sequence of steps. We assume every run satisfies the following conditions: (i) if a process is in Try, Critical, or Exit sections, it later executes a (normal or crash) step, and (ii) if a process enters Remainder because of a crash step, it later executes a (normal or crash) step.

An algorithm *solves* the RME problem if all of the following conditions are met in every run of the algorithm (Conditions (1), (3), (4) are from Golab and Ramaraju [78], and (2) and (5) are two additional natural conditions from Jayanti and Joshi [112]):

1. Mutual Exclusion: *At most one process is in the CS at any point.*

2. Wait-Free Exit: *There is a bound b such that, if a process π is in the Exit section, and executes steps without crashing, π completes the Exit section in at most b of its steps.*

3. Starvation Freedom: *If the total number of crashes in the run is finite and a process is in the Try section and does not subsequently crash, it later enters the CS.*

4. Critical Section Reentry (CSR) [78]: *If a process π crashes while in the CS, then no other process enters the CS during the interval from π's crash to the point in the run when π next reenters the CS.*

5. Wait-Free Critical Section Reentry (Wait-Free CSR) [112]: Given that the CSR property above mandates that after a process π's crash in the CS no other process may enter the CS until π reenters the CS, it makes sense to insist that no process should be able to obstruct π from reentering the CS. Specifically:

   There is a bound b such that, if a process crashes while in the CS, it reenters the CS before completing b consecutive steps without crashing.

   (As observed in [112], Wait-Free CSR, together with Mutual Exclusion, implies CSR.)

6.1.3 Passage Complexity

In a CC machine each process has a cache. A read operation by a process π on a shared variable X fetches a copy of X from shared memory to π's cache, if a copy is not already present. Any non-read operation on X by any process invalidates copies of X at all caches. An operation on X by π counts as a *remote memory reference* (RMR) if either the operation is not a read or X's copy is not in π's cache. When a process crashes, we assume that its cache contents are lost. In a DSM machine, instead of caches, shared memory is partitioned, with one partition residing at each process, and each shared variable resides in exactly one partition. Any operation (read or non-read) by a process on a shared variable X is counted as an RMR if X is not in π's partition.

A *passage* of a process π in a run starts when π enters Try (from Remainder) and ends at the earliest later time when π returns to Remainder (either because π crashes or because π completes Exit and moves back to Remainder).

A *super-passage* of a process π in a run starts when π either enters Try for the first time in the run or when π enters Try for the first time after the previous super-passage has ended, and it ends when π returns to Remainder by completing the Exit section.

The *passage complexity* (respectively, *super-passage complexity*) of an RME algorithm is the worst-case number of RMRs that a process incurs in a passage (respectively, in a super-passage).

6.1.4 Our contribution

The passage complexity of an RME algorithm can, in general, depend on n, the maximum number of processes the algorithm is designed for. The ideal of course would be to design an algorithm whose complexity is independent of n, but is this ideal achievable? It is well known that, for the traditional mutual exclusion problem, the answer is yes: MCS and many other algorithms that use FAS and CAS instructions have $O(1)$ passage complexity [41][56][152]. For the RME problem too, algorithms of $O(1)$ passage complexity are possible, but they use esoteric instructions not supported on real machines, such as Fetch-And-Store-And-Store (FASAS) and Double Word CAS, which manipulate two shared variables in a single atomic action [76][109]. The real question, however, is how well can we solve RME using only operations supported by real machines.

With their tournament based algorithm, Golab and Ramaraju showed that $O(\log n)$ passage complexity is possible using only read and write operations [78]. In fact, in light of Attiya et al's lower bound result [17], this logarithmic bound is the best that one can achieve even with the additional support of comparison-based operations such as CAS. However in PODC '17, by using FAS along with CAS, Golab and Hendler [76] succeeded in breaching this logarithmic barrier for CC machines: their algorithm has $O(\frac{\log n}{\log \log n})$ passage complexity for CC machines, but unbounded passage complexity for DSM machines. In this work, we close this gap with the design of an algorithm that achieves the same sub-logarithmic complexity bound as theirs for *both* CC and DSM machines. Some additional advantages of our algorithm over Golab and Hendler's are:

1. Our algorithm satisfies the Wait-Free Exit property.

2. On a CC machine, Golab and Hendler's algorithm requires a cache of $\Theta(n)$ words at each process, but our algorithm needs a cache of only $O(1)$ words. (We explain the reason in the next subsection.)

3. Our algorithm needs only the FAS instruction (whereas Golab and Hendler's needs both FAS and CAS).

4. Our algorithm eliminates the race conditions present in Golab and Hendler's algorithm that cause processes to starve.[1] (We describe these issues in detail in Appendix 6.A.)

[1] We communicated the issues described in Appendix 6.A with the authors of [76] who acknowledged the bugs and after a few weeks informed us that they were able to fix them.

6.1.5 Comparison to Golab and Hendler [76]: Similarities and differences

Golab and Hendler [76] derived their sublogarithmic RME algorithm in the following two steps, of which the first step is the intellectual workhorse:

- The first step is the design of an RME algorithm, henceforth referred to as GH, of $O(n)$ passage complexity and $O(1 + fn)$ super-passage complexity, where f is the number of times that a process crashes in the super-passage. The exciting implication of this result is that, in the common case where a process does not fail in a super-passage, the process incurs only $O(1)$ RMRs in the super-passage.
- The second step is the design of an RME algorithm where the n processes compete by working their way up on a tournament tree. This tree has n leaves and each of the tree nodes is implemented by an instance of GH in which $\log n / \log \log n$ processes compete. (thus, the degree of each node is $\log n / \log \log n$, which makes the tree's height $O(\log n / \log \log n)$). The resulting algorithm has the desired $O(\frac{\log n}{\log \log n})$ passage complexity and $O((1+f)\frac{\log n}{\log \log n})$ super-passage complexity.

The GH algorithm is designed by converting the standard MCS algorithm [152] into a recoverable algorithm. As we now explain, this conversion is challenging because MCS uses the FAS instruction to insert a new node at the end of a queue. The queue has one node for each process waiting to enter the CS, and a shared variable Tail points to the last node in the queue. When a process π enters the Try section, it inserts its node x into the queue by performing FAS(Tail, x). The FAS instruction stores the pointer to x in Tail and returns Tail's previous value *prev* to π in a single atomic action. The value in *prev* is vital because it points to x's predecessor in the queue. Suppose that π now crashes, thereby losing the *prev* pointer. Further suppose that a few more processes enter the Try section and insert their nodes behind π's node x. If π now restarts, it cannot distinguish whether it crashed just before performing the FAS instruction or just after performing it. In the former case, π will have to perform FAS to insert its node, but in the latter case it would be disastrous for π to perform FAS since x was already inserted into the queue. Yet, there appears no easy way for π to distinguish which of the scenarios it is in. Notice further that, like π, many other processes might have failed just before or after their FAS, causing the queue to be disconnected into several segments. All of these failed processes, upon restarting, have to go through the contents of the shared memory to recognize whether they are in the queue or not and,

if they are in, piece together their fragment with other fragments without introducing circularity or other blemishes in the queue. Since concurrent "repairing" by multiple recovering processes can lead to races, Golab and Hendler make the recovering processes go through an RME algorithm RLock so that at most one recovering process is doing the repair at any time. This RLock does not have to be too efficient since it is executed by only a failing process, which can afford to perform $O(n)$ RMRs. Thus, this RLock can be implemented using one of the known RME algorithms. However, while a process is trying to repair, correct processes can be constantly changing the queue, thereby making the repair task even more challenging.

The broad outline of our algorithm is the same as what we have described above for GH, but our algorithm differs substantially in important technical details, as we explain below.

- In GH a recovering process raises a FAIL flag only after confirming that there is evidence that its FAS was successful. This check causes GH to deadlock (see Scenario 1 in Appendix 6.A). Our algorithm eliminates this check.

- Since the shared memory can be constantly changing while a repairing process π is scanning the memory to compute the disjointed fragments (so as to connect π's fragment to another fragment), the precise order in which the memory contents are scanned can be crucial to algorithm's correctness. In fact, we found a race condition in GH that can lead to segments being incorrectly pieced together: two different nodes can end up with the same predecessor, leading to all processes starving from some point on (see Scenario 2 in Appendix 6.A).

- When a repairing process explores from each node x, GH does a "deep" exploration, meaning that the process visits x's predecessor x_1, x_1's predecessor x_2, x_2's predecessor x_3, and so on until the chain is exhausted. Our algorithm instead does a shallow exploration: it simply visits x's predecessor and stops there. The deep exploration of GH from each of the n nodes leads to $O(n^2)$ local computation steps per passage and requires each process to have a large cache of $O(n)$ words in order to ensure the desired $O(n)$ passage-RMR-complexity. With our shallow exploration, we reduce the number of local steps per passage to $O(n)$ and the RMR complexity of $O(n)$ is achieved with a cache size of only $O(1)$ words.

- How an exiting process hands off the ownership of CS to the next waiting process is done differently in our algorithm so as to ensure a wait-free Exit section and eliminate the need for the CAS instruction.

6.1.6 Related research

Beyond the works that we discussed above, Golab and Hendler [77] presented an algorithm at last year's PODC that has the ideal $O(1)$ passage complexity, but this result applies to a different model of system-wide crashes, where a crash means that *all* processes in the system *simultaneously* crash. Ramaraju [171] and Jayanti and Joshi [112] design RME algorithms that also satisfy the FCFS property [135]. These algorithms have $O(n)$ and $O(\log n)$ passage complexity, respectively. Attiya, Ben-Baruch, and Hendler present linearizable implementations of recoverable objects [15].

6.2 A Signal Object

Our main algorithm, presented in the next section, relies on a "Signal" object, which we specify and implement in this section. The Signal object is specified in Figure 6-1, which includes a description of two procedures — `set` and `wait` — through which the object is accessed.

X.State $\in \{1, 0\}$, initially 0.
- X.`set`() sets X.State to 1.
- X.`wait`() returns when X.State is 1.

Figure 6-1: Specification of a Signal object X.

6.2.1 An implementation of Signal Object

It is trivial to implement this object on a CC machine using a boolean variable Bit, initialized to 0. To execute `set`(), a process writes 1 in Bit, and to execute `wait`(), a process simply loops until Bit has 1. With this implementation, both operations incur just $O(1)$ RMRs on a CC machine. Realizing $O(1)$ RMR complexity on a DSM machine is less trivial, especially because the identity of the process executing `wait`() is unknown to the process executing `set`(). Figure 6-2 describes our DSM implementation $\mathcal{X}$ of a Signal object X, which assumes that no two processes execute the `wait`() operation concurrently on the Signal object. Our implementation provides two procedures: $\mathcal{X}$.`set`() and $\mathcal{X}$.`wait`(). Process π executes $\mathcal{X}$.`set`() to perform X.`set`() and $\mathcal{X}$.`wait`() to perform X.`wait`(). Our implementation ensures that a call to $\mathcal{X}$.`set`() and $\mathcal{X}$.`wait`() incur only $O(1)$ RMR.

When π invokes $\mathcal{X}$.`set`(), at Line **1** it records for future $\mathcal{X}$.`wait`() calls that X.State $=$ 1, hence those calls can return without waiting. Thereafter, π finds out if any process is already waiting for X.State to be set to 1. It does so by checking if any waiting process has supplied the address of its own local-spin variable to π on which it is waiting (Lines **2**-**3**). If π finds that a process is

Shared variables (stored in NVMM)
BIT $\in \{1, 0\}$, initially 0.
GOADDR is a **reference to** a **boolean**, initially NIL.

```
    procedure X.set()
1.  BIT ← 1
2.  addr_π ← GOADDR
3.  if addr_π ≠ NIL then
4.      *addr_π ← true
```

```
    procedure X.wait()
5.  go_π ← new Boolean
6.  *go_π ← false
7.  GOADDR ← go_π
8.  if BIT == 0 then
9.      wait till *go_π == true
```

Figure 6-2: Implementation of a Signal object specified in Figure 6-1. Code shown for a process π.

waiting (i.e., $addr_\pi \neq$ NIL), then it writes *true* into that process's spin-variable to wake it up from the wait loop (Line 4).

When a process π' invokes $\mathcal{X}$.wait(), at Line 5 it creates a new local-spin variable that it hosts in its own memory partition (Line 5). It initializes that variable for waiting (Line 6) and notifies the object about its address (Line 7) so that the caller of $\mathcal{X}$.wait() can wake π' up as described above. Then π' checks if BIT == 1 (Line 8), in which case X.STATE = 1 already and π' can return without waiting. Otherwise, π' waits for $*go_{\pi'}$ to turn *true* (Line 9).

Theorem 6.2.1. *$\mathcal{X}$.set() and $\mathcal{X}$.wait() described in Figure 6-2 implement a Signal object $\mathcal{X}$ (specified in Figure 6-1). Specifically, the implementation satisfies the following properties provided no two executions of $\mathcal{X}$.wait() are concurrent: (i) $\mathcal{X}$.set() is linearizable, i.e., there is a point in each execution of $\mathcal{X}$.set() when it appears to atomically set $\mathcal{X}$.STATE to 1, (ii) When $\mathcal{X}$.wait() returns, $\mathcal{X}$.STATE is 1, (iii) A process completes $\mathcal{X}$.set() in a bounded number of its own steps, (iv) Once $\mathcal{X}$.STATE becomes 1, any execution of $\mathcal{X}$.wait() by a process π completes in a bounded number of π's steps, (v) $\mathcal{X}$.set() and $\mathcal{X}$.wait() incur $O(1)$ RMR on each execution.*

6.3 The Algorithm

Our RME algorithm for k ports is presented in Figures 6-3-6-4. We assume that all shared variables are stored in non-volatile main memory, and process local variables (subscripted by π) are stored in respective processor registers. We assume that if a process uses a particular port during its super-passage in a run, then no other process will use the same port during that super-passage. The process decides the port it will use inside the Remainder section itself. Therefore, the algorithm presented in Figures 6-3-6-4 is designed for use by a process π on port p.

Types
QNode = **record**{PRED : **reference to** QNode,
NONNIL_SIGNAL : Signal object, CS_SIGNAL : Signal object } **end record**
Shared objects (stored in NVMM)
CRASH, INCS, and EXIT are distinct QNode instances, such that,
CRASH.PRED = &CRASH, INCS.PRED = &INCS, and EXIT.PRED = &EXIT.
SPECIALNODE is a QNode instance, such that, SPECIALNODE.PRED = &EXIT,
SPECIALNODE.NONNIL_SIGNAL = 1, and SPECIALNODE.CS_SIGNAL = 1.
RLOCK is a k-ported starvation-free RME algorithm
that incurs $O(k)$ RMR per passage on CC and DSM machines.
Shared variables (stored in NVMM)
TAIL is a **reference to** a QNode, initially &SPECIALNODE.
NODE is an **array**$[0 \ldots k-1]$ of **reference to** QNode. Initially, $\forall i$, NODE$[i]$ = NIL.

```
    Try Section
10. if NODE[p] = NIL then
11.     mynode_π ← new QNode
12.     NODE[p] ← mynode_π
13.     mypred_π ← FAS(TAIL, mynode_π)
14.     mynode_π.PRED ← mypred_π
15.     mynode_π.NONNIL_SIGNAL.set()
16. else
17.     mynode_π ← NODE[p]
18.     if mynode_π.PRED = NIL then mynode_π.PRED ← &CRASH
19.     mypred_π ← mynode_π.PRED
20.     if mypred_π = &INCS then go to Critical Section
21.     if mypred_π = &EXIT then
22.         Execute Lines 28-29 of Exit Section and go to Line 10
23.     mynode_π.NONNIL_SIGNAL.set()
24.     Execute RLOCK
25. mypred_π.CS_SIGNAL.wait()
26. mynode_π.PRED ← &INCS

    Exit Section
27. mynode_π.PRED ← &EXIT
28. mynode_π.CS_SIGNAL.set()
29. NODE[p] ← NIL
```

Figure 6-3: k-ported n-process RME algorithm for CC and DSM machines. Code shown for a process π that uses port $p \in \{0, \ldots, k-1\}$. (Code continued in Figure 6-4.)

6.3.1 Informal description

The symbol & is the usual "address of" operator, prefixed to a shared object to obtain the address of that shared object. The symbol "." (dot) dereferences a pointer and accesses a field from the record pointed to by that pointer. When invoked on a path σ in a graph, the functions `start`(σ) and `end`(σ) return the start and end vertices of the path σ. We assume that a process π is in the Remainder section when PC_π = **10** and is in the CS when PC_π = **27**.

Our algorithm uses a queue structure as in the MCS lock [152] and QNode is the node type used in such a queue. We modify the node structure in the following way to suit our needs. The node of a process π has, apart from a PRED pointer, two instances of a Signal object: CS_SIGNAL and NONNIL_SIGNAL. π's successor process will use the CS_SIGNAL instance from π's node to

Critical Section of RLock

30. **if** $mypred_\pi \neq$ &Crash **then go to** `Exit Section` of RLock
31. $tail_\pi \leftarrow$ Tail; $V_\pi \leftarrow \phi$; $E_\pi \leftarrow \phi$; $tailpath_\pi \leftarrow$ NIL; $headpath_\pi \leftarrow$ NIL
32. **for** $i_\pi \leftarrow 0$ **to** $k-1$
33. $\quad cur_\pi \leftarrow$ Node$[i_\pi]$
34. $\quad$ **if** $cur_\pi =$ NIL **then continue**
35. $\quad cur_\pi$.NonNil_Signal.`wait`()
36. $\quad curpred_\pi \leftarrow cur_\pi$.Pred
37. $\quad$ **if** $curpred_\pi \in$ {&Crash, &InCS, &Exit} **then** $V_\pi \leftarrow V_\pi \cup \{cur_\pi\}$
38. $\quad$ **else** $V_\pi \leftarrow V_\pi \cup \{cur_\pi, curpred_\pi\}$; $E_\pi \leftarrow E_\pi \cup \{(cur_\pi, curpred_\pi)\}$
39. Compute the set $Paths_\pi$ of maximal paths in the graph (V_π, E_π)
40. Let $mypath_\pi$ be the unique path in $Paths_\pi$ that contains $mynode_\pi$
41. **if** $tail_\pi \in V_\pi$ **then** let $tailpath_\pi$ be the unique path in $Paths_\pi$ that contains $tail_\pi$
42. **for each** $\sigma_\pi \in Paths_\pi$
43. $\quad$ **if** `end`(σ_π).Pred $\in$ {&InCS, &Exit} **then**
44. $\quad\quad$ **if** `start`(σ_π).Pred $\neq$ &Exit **then**
45. $\quad\quad\quad headpath_\pi \leftarrow \sigma_\pi$
46. **if** $tailpath_\pi =$ NIL $\vee$ `end`$(tailpath_\pi)$.Pred $\in$ {&InCS, &Exit} **then**
47. $\quad mypred_\pi \leftarrow$ **FAS**(Tail, `start`$(mypath_\pi)$)
48. **else if** $headpath_\pi \neq$ NIL **then** $mypred_\pi \leftarrow$ `start`$(headpath_\pi)$ **else** $mypred_\pi \leftarrow$ &SpecialNode
49. $mynode_\pi$.Pred $\leftarrow mypred_\pi$

Figure 6-4: (Code continued from Figure 6-3.) k-ported n-process recoverable mutual exclusion algorithm for CC and DSM machines. Code shown for a process π that uses port $p \in \{0, \ldots, k-1\}$. Vertex names in V_π are node references, hence the "." symbol dereferences the address and accesses the members of the node. The functions `start`(σ) and `end`(σ) used at Lines **43**, **44**, **46-48** return the start and end vertices of the path σ.

wait on π before entering the CS. The NonNil_Signal instance is used by any repairing process to wait till π sets the Pred pointer of its node to a value other than NIL. Every node has a unique instance of these Signal objects. We ensure that the call to CS_Signal.`wait`() happens from a single predecessor and the call to NonNil_Signal.`wait`() is made in a mutually exclusive manner, thus ensuring that no two executions of `wait`() are concurrent on the same object instance. We also use an array of references to QNodes called Node[]. This is a reference to a QNode that is used by some process on port p to complete a passage. In essence Node$[p]$ binds a process π to the port p through the QNode π uses for its passage.

We first describe how π would execute the Try and Exit section in absence of a crash as follows, and then proceed to explain the algorithm if a crash is encountered anywhere. When a process π wants to enter the CS through port p from the Remainder section, it starts executing the Try section. At Line **10** it checks if any previous passage ended in a crash. If that is not the case, π finds Node$[p]$ = NIL. It then executes Line **11** which allocates a new QNode for π in the NVMM, such that the Pred pointer holds NIL, and the objects CS_Signal and NonNil_Signal have State = `0` (i.e., their initial values). At Line **12** the process stores a reference to this new node in Node$[p]$ so that it can reuse this node in future in case of a crash. π then links itself to the

queue by swapping $mynode_\pi$ into TAIL (Line 13) and stores the previous value of TAIL (copied in $mypred_\pi$) into $mynode_\pi$.PRED (Line 14). The value of $mypred_\pi$, from Line 13 onwards, is an address of π's predecessor's node. At Line 15 π announces that it has completed inserting itself in the queue by setting $mynode_\pi$.NONNIL_SIGNAL to 1 (more later on why is this announcement important). π then proceeds to Line 25 where it waits for $mypred_\pi$.CS_SIGNAL to become 1. If the owner of the node pointed by $mypred_\pi$ has already left the CS, then $mypred_\pi$.CS_SIGNAL is 1; otherwise, π has to wait for a signal from its predecessor (see description of Signal object in previous section). Once π comes out of the call to $mypred_\pi$.CS_SIGNAL.wait(), it makes a note in $mynode_\pi$.PRED that it has ownership of the CS (Line 26). π then proceeds to the CS.

When π completes the CS, it first makes a note to itself that it no longer needs the CS by writing &EXIT in $mynode_\pi$.PRED (Line 27). It then wakes up any successor process that might be waiting on π to enter the CS (Line 28). π then writes NIL into NODE[p] at Line 29, which signifies that the passage that used this node has completed.

When π begins a passage after the previous passage ended in a crash, π starts by checking NODE[p] at Line 10. If it has the value NIL, then π crashed before it put itself in the queue, hence it treats the situation as if π didn't crash in the previous passage and continues as described above. Otherwise, π moves to Line 17 where it recovers the node it was using in the previous passage. If π crashed while putting itself in the queue (i.e., right before executing Lines 13 or 14), it treats the crash as if it performed the FAS at Line 13 and crashed immediately. Hence, it makes a note to itself that it crashed by writing &CRASH in $mynode_\pi$.PRED (Line 18). It then reads the value of $mynode_\pi$.PRED into $mypred_\pi$ (Line 19). At Line 20 π checks if it crashed while in the CS, in which case it moves to the CS. At Line 21 it checks if it already completed executing the CS, in which case recovery is done by executing Lines 28-29 and then re-executing Try from Line 10. If π reaches Line 23, it is clear that it crashed before entering the CS in the previous passage. In that case repairing the queue might be needed if π didn't set $mynode_\pi$.PRED to point to a predecessor node. In any case, π announces that $mynode_\pi$.PRED no longer has the value NIL setting $mynode_\pi$.NONNIL_SIGNAL to 1. π then goes on to capture RLOCK so that it gets exclusive access to repair the queue if it is broken at its node.

High level view of repairing the queue after a crash

Before diving into the code commentary of the CS of RLOCK, where π repairs the queue broken at its end, we describe how the repairing happens at a high level. π uses the RLOCK to repair

the queue, if it crashed around the FAS operation (Lines **13-14**) in the Try section. A crash by a process on Lines **13-14** can give rise to the following scenarios: (i) the queue is not affected by the crash (crash at Line **13** or at Line **14** but the queue was already broken), (ii) the queue is broken due to the crash (crash at Line **14**). Therefore consider the following configuration[2]. Assume there is a node x that was used by some process in its passage and the process has completed that passage succesfully so that x.PRED = &EXIT. Process π_1, π_3, and π_5 have crashed at Line **14**. Process π_2, π_4, and π_6 are executing the procedure `wait()` at Line **25**, such that, π_2's predecessor is π_1, π_4's predecessor is π_3, and π_6's predecessor is π_5. Process π_7 and π_8 have crashed at Line **13**. We describe the repair by each of these crashed processes as follows.

Each of the crashed processes executes the RLOCK and waits for its turn to repair the queue in a mutually exclusive manner. Assume that the repair is performed by the processes in the order: $(\pi_1, \pi_7, \pi_5, \pi_8, \pi_3)$. When π_1 performs the repair, it first scans the NODE array and notices that the queue is broken at process π_4 and π_5's nodes (it notices that by reading &CRASH in the PRED pointer of the process nodes). NODE array also gives an illusion to π_1 that queue is broken at π_7 and π_8's node although these processes didn't perform a FAS prior to their crash. π_1 also notices that no node has a predecessor node whose PRED pointer is set to &INCS or &EXIT, hence, no process is in the CS or is poised to enter it. Therefore π_1 sets its own node's predecessor to be SPECIALNODE (from Figure 6-3, SPECIALNODE.PRED = &EXIT). Note, no other crashed process will set their own node's PRED pointer to point to SPECIALNODE simultaneously because repair operation is performed in a mutually exclusive manner by π_1. Also, the PRED pointer of each node has a non-NIL value (if not, then π_1 waits till it sees a non-NIL value before doing the actual repair). This way π_1 completes the repair operation on the queue and is now poised to enter the CS.

When π_7 (crashed at Line **13**) performs the repair, it first scans the NODE array and notices that the queue is broken at process π_4, π_5, π_7, and π_8's nodes. Since it notices that no process points to π_2, it sets the PRED pointer of its own node to point to π_2's node. Thereby π_7 finishes the repair by placing itself in the queue, without ever performing the FAS, and gives up its control over RLOCK to return to the Try section.

When π_5 (crashed at Line **19**) performs the repair, it follows an approach similar to that of π_7's. It sees that the queue is broken at process π_3, π_5, and π_8. It then notices that no process points to π_7 and therefore sets the PRED pointer of its own node to point to π_7's node. This way π_5 and π_6

[2]Please refer to Figure 6.B.1 of Section 6.B in the Appendix for a visual illustration.

are now attached to the queue in a way that there is a path from their node to a node containing the address &Exit. Also, Tail points to π_6's node, so it appears as if the queue is unbroken if a traversal was done starting at the Tail pointer.

Now that a traversal from Tail would lead to a node used by a process that is in Critical section (π_1 in this case), the queue is partially in place. In order to fix the remaining broken fragments the queue might need to be broken somehow to fit the remaining fragments. However, π_3 and π_8 can do the repair from here on without affecting the existing structure of the queue. π_8 can put itself in the queue by performing the FAS operation on the Tail with its own node. Whereas π_3 first identifies the fragment its node is part of, and thereby all the nodes that are part of its fragment. It then performs a FAS one more time on Tail with the last node in its own fragment (i.e., π_4's node) and sets the Pred pointer of its own node to the previous value of Tail that is returned by the FAS (address of π_8's node). This ends the repair operation for π_3 and thereby the repair for all the process.

Informal description of CS of RLock

We proceed to give a description of the CS of RLock that does the above mentioned repair. At Line **30** π checks if it was already in the queue before its last crash (such a situation may occur either when π crashes after executing the CS of RLock to completion but before executing the Exit section of RLock, or when π crashes in the Try after performing Line **14**). If so, it notices that there is no need for repair, hence, it goes to the Exit section of RLock. Otherwise, at Line **31** π reads the reference to the node pointed to by Tail into the variable $tail_\pi$ and initializes other variables used during the repair procedure. Thereafter π constructs a graph that models the queue structure. To this purpose, it reads each node pointed to by the Node array in order to construct the graph (Lines **32**-**38**). The graph is constructed as follows. First a cell from the Node array is read into cur_π (i.e., Node$[i_\pi]$) at Line **33** and checked if it is a node of some process (Line **34**). If Node$[i_\pi]$ = NIL, π moves on to the next cell in the array. Otherwise, at Line **35** π waits till Node$[i_\pi]$.Pred assumes a non-NIL value (i.e., wait for the owner of that node to have executed either Line **15** or **23**). Once cur_π's Pred pointer has a non-NIL value, that value is read into $curpred_\pi$ (Line **36**). There are now two possibilities: (i) the Pred pointer points to one of &Crash, &InCS, or &Exit, or (ii) the Pred pointer points to another node. The purpose of waiting for cur_π.NonNil_Signal = `1` is simple: we want to be sure which of the above two cases is true about cur_π. In the first case only cur_π is added as a vertex to the graph (the name

of that vertex is the value of cur_π). In the second case cur_π and $curpred_\pi$ are added as vertices and a directed edge $(cur_\pi, curpred_\pi)$ is added to the graph (we consider this as a simple graph, so repeated addition of a vertex counts as adding it once). This process is repeated until all cells from the NODE array are read. Once all the nodes are read from the cells of NODE array, including nodes not yet in the queue (π_7 and π_8 in the above example), we have the graph (V_π, E_π) that models the broken queue structure such that each maximal path in the graph models a broken queue fragment. Note, such a graph is a directed acyclic graph. At Line **39** set $Paths_\pi$ of maximal paths in the graph (V_π, E_π) is created and at Line **40** a path $mypath_\pi$ is picked from $Paths_\pi$ such that $mynode_\pi$ appears in it. At Line **41** a path $tailpath_\pi$ containing the node $tail_\pi$ is picked from $Paths_\pi$ if $tail_\pi$ appears in the graph. In Lines **42-45** we try to find a path in the graph such that its start vertex belongs to a process that has not finished the critical section but a traversal on that path leads to a node holding one of the addresses &INCS or &EXIT (i.e., it leads to a node in or out of CS). If such a path is found, $headpath_\pi$ is set to point to that path, otherwise, $headpath_\pi$ remains NIL. In Line **46** we first check if the queue is already partially repaired (e.g., if the repair was being performed by π_8 or π_3 in the example above). If so, at Line **47** the fragment containing $mynode_\pi$ is inserted into the queue by performing a FAS on TAIL with the last node in that fragment (i.e., `start`$(mypath_\pi)$ would give the address of last node appearing in $mynode_\pi$'s fragment). We note the previous value of TAIL into $mypred_\pi$ so that we can update $mynode_\pi$.PRED later. Otherwise, π needs to connect its own fragment to the queue. To this purpose it needs to be ensured that the queue is not broken at its head and some active process is poised to enter or is in the Critical section. Line **48** does this by checking if Lines **42-45** found a path in the graph such that its start vertex belongs to a process that has not finished the Critical section but a traversal on that path leads to a node out of CS (i.e., is $headpath_\pi \neq$ NIL). If $headpath_\pi \neq$ NIL, then π's predecessor is set to be the start node on the path $headpath_\pi$ (π_7, π_5 in the example above). Otherwise, the queue is broken at its head, therefore, at Line **48**, π's predecessor is set to be SPECIALNODE (π_1 in example above). At Line **49**, π has the correct address to its predecessor node in $mypred_\pi$ (as noted in Lines **46-48**) which is written into $mynode_\pi$.PRED. This completes the CS of RLOCK and the repair of π's fragment. π then proceeds back to Line **25** after completing the Exit section of RLOCK.

6.3.2 Main theorem

The correctness properties of the algorithm are captured in the following theorem.

Theorem 6.3.1. *The algorithm in Figures 6-3-6-4 solves the RME problem for k ports on CC and DSM machines and additionally satisfies the Wait-free Exit and Wait-free CSR properties. It has an RMR complexity of $O(1)$ for a process that does not crash during its passage, and $O(fk)$ for a process that crashes f times during its super-passage.*

6.3.3 $O((1+f)\log n/\log\log n)$ RMRs Algorithm

To obtain a sub-logarithmic RMR complexity algorithm on both CC and DSM machines, we use the arbitration tree technique used by Golab and Hendler (described in Section 5 in [76]). Therefore, the following theorem follows from Theorem 6.3.1.

Theorem 6.3.2. *The arbitration tree algorithm solves the RME problem for n processes on CC and DSM machines and additionally satisfies the Wait-free Exit and Wait-free CSR properties. It has an RMR complexity of $O((1+f)\log n/\log\log n)$ per super-passage for a process that crashes f times during its super-passage.*

Appendix

6.A Issues with Golab and Hendler's [76] Algorithm

In this section we describe two issues with Golab and Hendler's FAS and CAS based algorithm. The Algorithm in question here appears in Figures 6, 7, 8 in [76] and we use the exact line numbers and variable names as they appear in the thesis.

6.A.1 Scenario 1: Process deadlock inside Recover

The first issue with the GH algorithm is that processes deadlock waiting on each other inside the Recover section. This issue is described as below:

1. Process P_4 requests the lock by starting a fresh passage, goes to the CS, completes the Exit, and then goes back to Remainder.

2. Process P_2 starts a fresh passage, executes the code till (but not including) Line 26 and crashes.

3. Remainder section puts P_2 into Recover, P_2 starts executing `IsLinkedTo(2)` from Line 44 because $mynode.nextStep = 26$ and $mynode.prev = \perp$ for P_2.

4. P_2 sleeps at Line 68 with $i = 0$.

5. Process P_4 starts another passage, executes till (but not including) Line 26 and crashes.

6. Thereafter, P_4 goes to Recover, starts executing `IsLinkedTo(4)` from Line 44 because $mynode.nextStep = 26$ and $mynode.prev = \perp$ for P_4.

7. P_2 starts executing procedure `IsLinkedTo()` where it left and executes several interations until $i = 4$. Now it waits on $lnodes[4].prev$ (P_4's $mynode$) to become non-$\perp$.

8. P_4 starts executing procedure `IsLinkedTo()` where it left and executes several iterations until $i = 2$ and now it waits on $lnodes[2].prev$ (P_2's $mynode$) to become non-$\perp$.

9. From now on no process including P_2 and P_4 ever crash. Therefore P_2 and P_4 are then waiting on each other and no one ever sets $mynode.prev$ to a non-$\perp$ value. This results in violation of Starvation freedom property.

6.A.2 Scenario 2: Starvation Freedom Violation

The second issue with their algorithm is a process may starve even though it never crashed. The issue is as described below:

1. Process P_0 initiates a new passage, goes to CS, and no other process comes after it, so $tail$ is pointing to P_0's node.

2. P_1 initiates a new passage, performs FAS on $tail$ and goes behind P_0, and sets its own $mynode.prev$ field to point to P_0's node.

3. P_2 initiates a new passage, performs FAS on $tail$ and goes behind P_1, but crashes immediately, hence losing its local variable $prev$ before setting its own $mynode.prev$ field.

4. P_2 performs `isLinkedTo(2)`, which returns $true$ because $tail$ is pointing to P_2's $mynode$.

5. P_3 initiates a new passage, performs FAS on $tail$ and goes behind P_2, and sets its own $mynode.prev$ field to point to P_2's node.

6. P_2 acquires $rLock$ in order to recover from the crash, and performs iterations with $i = 0, 1, 2, 3$ of the for-loop on Line 76. At this point the relation R maintained in the $rlock$ contains (0, 1), (2, 3), (3, TAIL).

7. P_4 initiates a new passage, performs FAS on $tail$ and goes behind P_3, but loses its local variable $prev$ before setting its own $mynode.prev$ field.

8. P_5 initiates a new passage, performs FAS on $tail$ and goes behind P_4, and sets its own $mynode.prev$ field to point to P_4's node.

9. P_2 resumes and performs iterations with $i = 4, 5$ of the for-loop at Line 76, adding (4,5) to R.

At this point R = (0, 1), (2, 3), (3, TAIL), (4,5). Therefore, process 2 identifies

- (0,1) as the non-failed fragment (segment 1),
- (4,5) as the middle segment (segment 2), and
- (2,3), (3,TAIL) as the tail segment (segment 3).

10. On Line 93 P_2 sets *mynode.prev* to point to P_5's node and *tail* still points to P_5's node.

11. P_6 initiates a new passage, performs FAS on *tail* and goes behind P_5, and sets its own *mynode.prev* field to point to P_5's node. Note, at this point, both P_2 and P_6 set their respective *mynode.prev* field to point to the P_5's node and *tail* points to P_6's node.

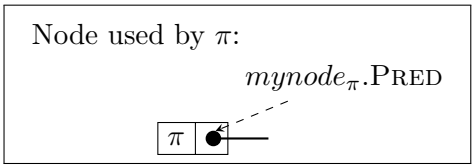
12. Thereafter P_6 executes the remaining lines of Try section setting P_5's *mynode.next* to point to its own node at Line 30, and then continues to busy-wait on Line 31.

13. P_2 then comes out of the *rlock*, continues to Line 28 in Try, sets P_5's *mynode.next* to point to its own node at Line 30, and continues to busy-wait on Line 31.

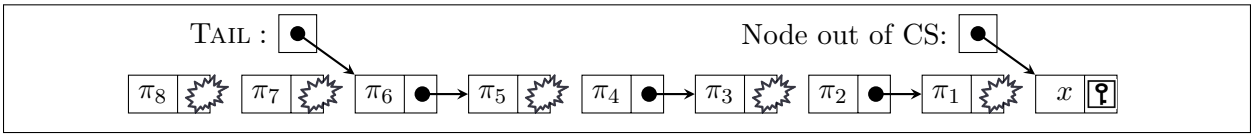
14. Hereafter, assume that no process fails, we have that all the processes coming after P_6 including P_6 itself forever starve. This is because P_5 was supposed to wake P_6 up from the busy-wait, but it would wake up P_2 instead. P_2 never wakes any process up because it is not visible to any process. This violates Starvation Freedom.

6.B Illustration for Repair

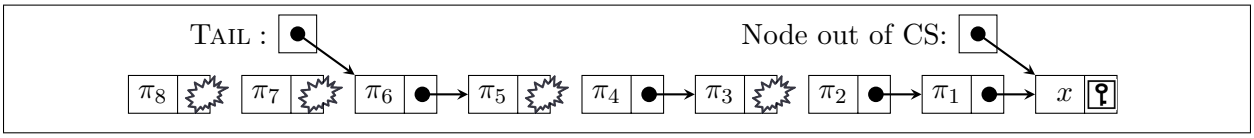
Figure 6.B.1 illustrates the bird's eye view of queue repair performed by crashed processes. Refer to Section 6.3.1 for a detailed description.



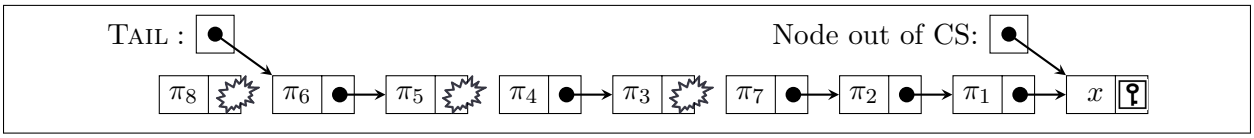
With π_1, π_3, π_5, π_7, and π_8 crashed, initial state of the queue (π_1, π_3, π_5 crashed at Line **14** and π_7, π_8 crashed at Line **13**):



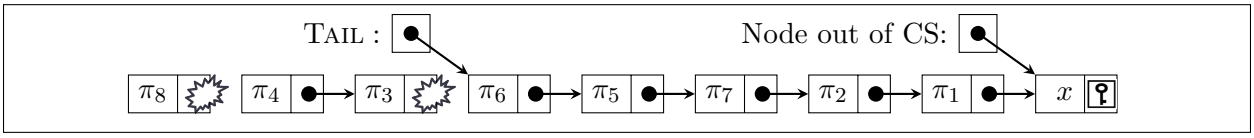
π_1 performs repair:



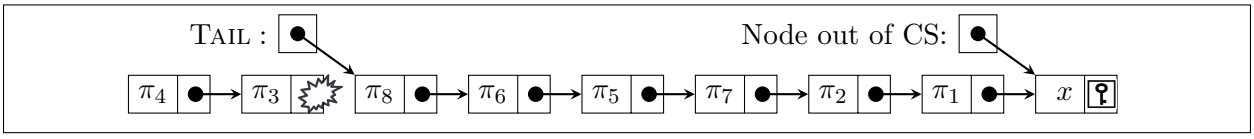
π_7 performs repair:



π_5 performs repair:



π_8 performs repair:



π_3 performs repair:

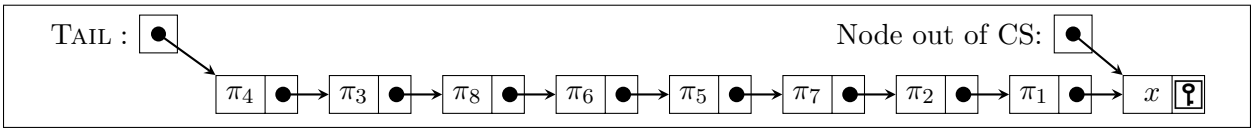


Figure 6.B.1: Queue states after repair is performed by different processes in a sequence. Explosion symbol in place of a Pred pointer on a node denotes the said process has crashed without updating the Pred pointer of its node.

6.C Proof of correctness

In this section we present a proof of correctness for the algorithm presented in Figures 6-3-6-4. We prove the algorithm by giving an invariant for the algorithm and then proving correctness using the invariant. Figures 6.C.3-6.C.6 give the invariant satisfied by the algorithm. The proof that the algorithm satisfies the invariant is by induction and is presented in Appendix 6.E.

We begin with some notation used in the proof and the invariant. A process may crash several times during its super-passage, at which point all its local variables get wiped out and the program counter is reset to **10** (i.e. first instruction of Try). In order to prove correctness we maintain a set of hidden variables that help us in the arguments of our proof. Following is the list of hidden variables for a process π and the locations that the variables are updated in the algorithm:

$\widehat{port_\pi}$: This variable stores the port number that π uses to complete its super-passage. The Remainder section decides which port will be used by π for the super-passage. When π is not active in a super-passage, we assume that $\widehat{port_\pi} = \text{NIL}$.

$\widehat{PC_\pi}$: This variable takes line numbers as value according to the value of program counter, i.e., PC_π. Figures 6.C.1-6.C.2 show the annotated versions of our code from Figures 6-3-6-4 (annotations in <>) where we show the value that $\widehat{PC_\pi}$ takes at each line. We assume that the change in $\widehat{PC_\pi}$ happens atomically along with the execution of the line. $\widehat{PC_\pi}$ remains the same as before a line is executed for those lines in the figure that are not annotated (for example, Lines **10**, **16**-**21**).

$\widehat{node_\pi}$: This variable is used to denote the QNode that π is using in the current configuration for the current passage. Detailed description of the values that $\widehat{node_\pi}$ takes appears in the Definitions section of Figure 6.C.3.

```
    Try Section
10. if Node[p] = NIL then
11.     mynode_π ← new QNode; < PC_π ← 12 >
12.     Node[p] ← mynode_π; < PC_π ← 13 >
13.     mypred_π ← FAS(Tail, mynode_π); < PC_π ← 14 >
14.     mynode_π.Pred ← mypred_π; < PC_π ← 15 >
15.     mynode_π.NonNil_Signal.set(); < PC_π ← 25 >
16. else
17.     mynode_π ← Node[p]
18.     if mynode_π.Pred = NIL then mynode_π.Pred ← &Crash
19.     mypred_π ← mynode_π.Pred
20.     if mypred_π = &InCS then go to Critical Section
21.     if mypred_π = &Exit then
22.         Execute Lines 28-29 of Exit Section and go to Line 10; < PC_π ← 11 >
23.     mynode_π.NonNil_Signal.set()
24.     Execute RLock
25. mypred_π.CS_Signal.wait(); < PC_π ← 26 >
26. mynode_π.Pred ← &InCS; < PC_π ← 27 >

    Exit Section
27. mynode_π.Pred ← &Exit; < PC_π ← 28 >
28. mynode_π.CS_Signal.set(); < PC_π ← 29 >
29. Node[p] ← NIL; < PC_π ← 11 >
```

Figure 6.C.1: Annotated version of code from Figure 6-3. $\widehat{port_\pi} = p$.

```
    Critical Section of RLock
30. if mypred_π ≠ &Crash then
        go to Exit Section of RLock;
        < PC_π ← 25 >
31. tail_π ← Tail; V_π ← φ; E_π ← φ; tailpath_π ← NIL; headpath_π ← NIL
32. for i_π ← 0 to k − 1
33.     cur_π ← Node[i_π]
34.     if cur_π = NIL then continue
35.     cur_π.NonNil_Signal.wait()
36.     curpred_π ← cur_π.Pred
37.     if curpred_π ∈ {&Crash, &InCS, &Exit} then V_π ← V_π ∪ {cur_π}
38.     else V_π ← V_π ∪ {cur_π, curpred_π}; E_π ← E_π ∪ {(cur_π, curpred_π)}
39. Compute the set Paths_π of maximal paths in the graph (V_π, E_π)
40. Let mypath_π be the unique path in Paths_π that contains mynode_π
41. if tail_π ∈ V_π then let tailpath_π be the unique path in Paths_π that contains tail_π
42. for each σ_π ∈ Paths_π
43.     if end(σ_π).Pred ∈ {&InCS, &Exit} then
44.         if start(σ_π).Pred ≠ &Exit then
45.             headpath_π ← σ_π
46. if tailpath_π = NIL ∨ end(tailpath_π).Pred ∈ {&InCS, &Exit} then
47.     mypred_π ← FAS(Tail, start(mypath_π)); < PC_π ← 14 >
48. else
        if headpath_π ≠ NIL then mypred_π ← start(headpath_π) else mypred_π ← &SpecialNode ;
        < PC_π ← 14 >
49. mynode_π.Pred ← mypred_π ; < PC_π ← 25 >
```

Figure 6.C.2: Annotated version of code from Figure 6-4. $\widehat{port_\pi} = p$.

We say that a process is in the CS if and only if $\widehat{PC_\pi} = \mathbf{27}$. If π is not active in a super-passage and hence in the Remainder section, $PC_\pi = \mathbf{10}$, $\widehat{PC_\pi} = \mathbf{15}$, and the values of the rest of the hidden variables are as defined above. We assume that initially all the local variables take arbitrary values.

Assumptions:

- Algorithm in Figures 6-3-6-4 assumes that every process uses a single port throughout its super-passage and no two processes execute a super-passage with the same port when their super-passages overlap. The Remainder section ensures that this assumption is always satisfied. Therefore, when a process continues execution after a crash, it uses the same port it chose at the start of the current super-passage. Hence, the Remainder section guarantees that the following condition is always met for active processes:
  $\forall \pi \in \Pi, \exists p \in \mathcal{P}, (\widehat{port_\pi} = p \wedge \forall p' \in \mathcal{P}, p \neq p') \Rightarrow \widehat{port_\pi} \neq p'$.

Definitions (Continued in Figure 6.C.4):

- $\mathcal{P}$ is a set of all ports.
- Π is a set of all processes.
- $\mathcal{N}$ is a set containing the node SpecialNode and any of the QNodes created by any process at Line **11** during the run so far.
- $\mathcal{N}' = \{\&qnode \mid qnode \in \mathcal{N}\}$ is a set of node addresses from the nodes in $\mathcal{N}$.
- $\widehat{node_\pi} = \begin{cases} \text{Node}[\widehat{port_\pi}], & \text{if } \widehat{PC_\pi} \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{29}], \\ mynode_\pi, & \text{if } PC_\pi = \mathbf{12}, \\ \text{NIL}, & \text{otherwise (i.e., } \widehat{PC_\pi} \in [\mathbf{11}, \mathbf{12}] \wedge PC_\pi \in [\mathbf{10}, \mathbf{11}]). \end{cases}$

Conditions (Continued in Figures 6.C.4-6.C.5):

1. $\forall \pi \in \Pi, (\widehat{PC_\pi} \in \{\mathbf{11}, \mathbf{12}\} \Leftrightarrow \text{Node}[\widehat{port_\pi}] = \text{NIL}) \wedge (\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\} \Leftrightarrow \widehat{node_\pi}.\text{Pred} \in \{\text{NIL}, \&\text{Crash}\})$
   $\wedge (\widehat{PC_\pi} \in \{\mathbf{15}, \mathbf{25}, \mathbf{26}\} \Leftrightarrow \widehat{node_\pi}.\text{Pred} \in \mathcal{N}') \wedge (\widehat{PC_\pi} = \mathbf{27} \Leftrightarrow \widehat{node_\pi}.\text{Pred} = \&\text{InCS})$
   $\wedge (\widehat{PC_\pi} \in \{\mathbf{28}, \mathbf{29}\} \Leftrightarrow \widehat{node_\pi}.\text{Pred} = \&\text{Exit})$
2. $\forall \pi \in \Pi, (PC_\pi \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{18}, \mathbf{29}] \cup [\mathbf{30}, \mathbf{48}] \Rightarrow mynode_\pi = \text{Node}[\widehat{port_\pi}])$
   $\wedge (PC_\pi \in \{\mathbf{15}\} \cup [\mathbf{20}, \mathbf{24}] \cup [\mathbf{25}, \mathbf{26}] \cup [\mathbf{30}, \mathbf{48}] \Rightarrow mypred_\pi = \text{Node}[\widehat{port_\pi}].\text{Pred})$
   $\wedge ((PC_\pi \in [\mathbf{20}, \mathbf{24}] \cup [\mathbf{30}, \mathbf{48}] \wedge \widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\}) \Rightarrow mypred_\pi = \&\text{Crash})$
3. $\forall \pi \in \Pi, \text{Node}[\widehat{port_\pi}] \neq \text{NIL} \Rightarrow (\text{Node}[\widehat{port_\pi}] \in \mathcal{N}'$
   $\wedge ((\exists \pi' \in \Pi, \pi \neq \pi' \wedge \text{Node}[\widehat{port_\pi}].\text{Pred} = \text{Node}[\widehat{port_{\pi'}}])$
   $\vee (\text{Node}[\widehat{port_\pi}].\text{Pred} \in \mathcal{N}' \wedge \text{Node}[\widehat{port_\pi}].\text{Pred}.\text{Pred} = \&\text{Exit})$
   $\vee \text{Node}[\widehat{port_\pi}].\text{Pred} \in \{\text{NIL}, \&\text{Crash}, \&\text{InCS}, \&\text{Exit}\})$
   $\wedge (\forall \pi'' \in \Pi, \pi \neq \pi'' \Rightarrow$
   $((\text{Node}[\widehat{port_\pi}] = \text{Node}[\widehat{port_{\pi''}}] \Rightarrow \text{Node}[\widehat{port_\pi}] = \text{NIL})$
   $\wedge (\text{Node}[\widehat{port_\pi}].\text{Pred} = \text{Node}[\widehat{port_{\pi''}}].\text{Pred} \Rightarrow$
   $\text{Node}[\widehat{port_\pi}].\text{Pred} \in \{\text{NIL}, \&\text{Crash}, \&\text{Exit}\})))))$
4. $\forall \pi, \pi' \in \Pi, (\pi \neq \pi' \Rightarrow (\widehat{node_\pi} \neq \widehat{node_{\pi'}} \vee \widehat{node_\pi} = \widehat{node_{\pi'}} = \text{NIL}))$
   $\wedge ((\pi \neq \pi' \wedge \widehat{node_\pi} \neq \text{NIL} \wedge \widehat{node_{\pi'}} \neq \text{NIL}) \Rightarrow$
   $(\widehat{node_\pi}.\text{Pred} \neq \widehat{node_{\pi'}}.\text{Pred} \vee \widehat{node_\pi}.\text{Pred} \in \{\text{NIL}, \&\text{Crash}, \&\text{Exit}\}))$
   $\wedge (\exists b \in \mathbb{N}, (1 \leq b \leq k \wedge \widehat{node_\pi}.\underbrace{\text{Pred}.\text{Pred} \cdots .\text{Pred}}_{b \text{ times}} \in \{\text{NIL}, \&\text{Crash}, \&\text{InCS}, \&\text{Exit}\}))$
5. $\forall\ qnode \in \mathcal{N}, qnode.\text{Pred} \in \{\text{NIL}, \&\text{Crash}, \&\text{InCS}, \&\text{Exit}\} \cup \mathcal{N}'$
   $\wedge ((\forall \pi \in \Pi, \widehat{node_\pi} \neq \&qnode) \Leftrightarrow (\forall p' \in \mathcal{P}, \text{Node}[p] \neq qnode \wedge \forall \pi' \in \Pi, mynode_{\pi'} \neq \&qnode))$
   $\wedge (qnode.\text{CS\_Signal} = \texttt{1} \Rightarrow (qnode.\text{Pred} = \&\text{Exit} \wedge (\forall \pi \in \Pi, \widehat{node_\pi} = qnode \Rightarrow \widehat{PC_\pi} = \mathbf{29})))$
   $\wedge (qnode.\text{NonNil\_Signal} = \texttt{1} \Rightarrow$
   $(qnode.\text{Pred} \neq \text{NIL} \wedge (\forall \pi \in \Pi, \widehat{node_\pi} = qnode \Rightarrow \widehat{PC_\pi} \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{29}])))$
   $\wedge (qnode.\text{CS\_Signal} = \texttt{0} \Rightarrow qnode.\text{Pred} \in \{\text{NIL}, \&\text{Crash}, \&\text{InCS}\})$
   $\wedge (qnode.\text{NonNil\_Signal} = \texttt{0} \Rightarrow qnode.\text{Pred} = \text{NIL})$

Figure 6.C.3: Invariant for the k-ported recoverable mutual exclusion algorithm from Figures 6-3-6-4. (Continued in Figures 6.C.4-6.C.5.)

Definitions (Continued from Figure 6.C.3):

- For a QNode instance $\widehat{node}_\pi$ used by a process $\pi \in \Pi$, $\texttt{fragment}(\widehat{node}_\pi)$ is a sequence of distinct QNode instances $(\widehat{node}_{\pi_1}, \widehat{node}_{\pi_2}, \ldots, \widehat{node}_{\pi_j})$ such that:
  - $\forall i, \widehat{node}_{\pi_i} \in \mathcal{N}$,
  - $\forall i \in [1, j-1], \widehat{node}_{\pi_{i+1}}.\text{PRED} = \widehat{node}_{\pi_i}$ (e.g., $\widehat{node}_{\pi_2}.\text{PRED} = \widehat{node}_{\pi_1}$),
  - $\widehat{node}_{\pi_1}.\text{PRED} \in \{\text{NIL}, \&\text{CRASH}, \&\text{INCS}, \&\text{EXIT}\}$,
  - $\forall q \in \mathcal{P}, \text{NODE}[q].\text{PRED} \neq \widehat{node}_{\pi_j}$,
  - $\texttt{head}(\texttt{fragment}(\widehat{node}_\pi)) = \widehat{node}_{\pi_1}$ and $\texttt{tail}(\texttt{fragment}(\widehat{node}_\pi)) = \widehat{node}_{\pi_j}$,
  - $|\texttt{fragment}(\widehat{node}_\pi)| = j$.

  For example, for the initial state of the queue in Figure 6.B.1, (π_1, π_2), (π_3, π_4), (π_5, π_6), (π_7), (π_8) are distinct fragments. After π_3 performs repair in the illustration of Figure 6.B.1, the only fragment of the queue is: $(\pi_1, \pi_2, \pi_7, \pi_5, \pi_6, \pi_8, \pi_3, \pi_4)$. Note, in this example a node assumes the name of its process for brevity (i.e., π_1 should be read as $\widehat{node}_{\pi_1}$). The set membership symbol $\in$ used on the sequence denotes membership of a node in the fragment. For example, $\pi_2 \in \texttt{fragment}(\pi_1)$ in both examples discussed above. Note, for simplicity we define $\texttt{fragment}(\text{NIL}) = \text{NIL}$ and $|\texttt{fragment}(\text{NIL})| = 0$. Conditions of the invariant assert that the set of nodes in shared memory operated by the algorithm satisfy this definition of $\texttt{fragment}$.
- $\mathcal{Q} = \{\pi \in \Pi \mid (\widehat{PC}_\pi \in \{\mathbf{15}, \mathbf{25}, \mathbf{26}\} \wedge \texttt{head}(\texttt{fragment}(\widehat{node}_\pi)).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}) \vee \widehat{PC}_\pi = \mathbf{27}\}$ is a set of queued processes.

Conditions (Continued from Figure 6.C.3):

6. $\forall \pi \in \Pi, (PC_\pi = \mathbf{10} \Rightarrow \widehat{PC}_\pi \in [\mathbf{11}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{29}]) \wedge (PC_\pi = \mathbf{11} \Rightarrow \widehat{PC}_\pi \in [\mathbf{11}, \mathbf{12}])$
   $\wedge\ (PC_\pi \in [\mathbf{12}, \mathbf{15}] \cup \{\mathbf{25}\} \Rightarrow \widehat{PC}_\pi = PC_\pi) \wedge (PC_\pi \in [\mathbf{16}, \mathbf{20}] \Rightarrow \widehat{PC}_\pi \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{29}])$
   $\wedge\ (PC_\pi = \mathbf{21} \Rightarrow \widehat{PC}_\pi \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{26}] \cup [\mathbf{28}, \mathbf{29}]) \wedge (PC_\pi = \mathbf{22} \Rightarrow \widehat{PC}_\pi \in [\mathbf{28}, \mathbf{29}])$
   $\wedge\ (PC_\pi \in [\mathbf{23}, \mathbf{24}] \cup \{\mathbf{30}\} \Rightarrow \widehat{PC}_\pi \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{26}]) \wedge (PC_\pi \in [\mathbf{31}, \mathbf{48}] \Rightarrow \widehat{PC}_\pi \in [\mathbf{13}, \mathbf{14}])$
   $\wedge\ (\widehat{PC}_\pi = \mathbf{11} \Rightarrow PC_\pi \in \{\mathbf{10}, \mathbf{11}\}) \wedge (\widehat{PC}_\pi = \mathbf{12} \Rightarrow PC_\pi \in [\mathbf{10}, \mathbf{12}])$
   $\wedge\ (\widehat{PC}_\pi \in \{\mathbf{13}, \mathbf{14}\} \Rightarrow (PC_\pi = \widehat{PC}_\pi \vee PC_\pi \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{21}] \cup \{\mathbf{23}, \mathbf{24}\} \cup [\mathbf{30}, \mathbf{48}]))$
   $\wedge\ (\widehat{PC}_\pi \in \{\mathbf{15}, \mathbf{25}, \mathbf{26}\} \Rightarrow (PC_\pi = \widehat{PC}_\pi \vee PC_\pi \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{21}] \cup \{\mathbf{23}, \mathbf{24}\} \cup \{\mathbf{30}\}))$
   $\wedge\ (\widehat{PC}_\pi = \mathbf{27} \Rightarrow (PC_\pi = \widehat{PC}_\pi \vee PC_\pi \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{20}]))$
   $\wedge\ (\widehat{PC}_\pi \in \{\mathbf{28}, \mathbf{29}\} \Rightarrow (PC_\pi = \widehat{PC}_\pi \vee PC_\pi \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{22}]))$
7. $\forall \pi, \pi' \in \Pi, \texttt{fragment}(\widehat{node}_\pi) \neq \texttt{fragment}(\widehat{node}_{\pi'}) \Rightarrow$
   $(\forall \pi'' \in \Pi, \widehat{node}_{\pi''} \in \texttt{fragment}(\widehat{node}_\pi) \Rightarrow \widehat{node}_{\pi''} \notin \texttt{fragment}(\widehat{node}_{\pi'}))$
   $\wedge\ \texttt{head}(\texttt{fragment}(\widehat{node}_\pi)).\text{PRED} = \&\text{INCS} \Rightarrow ((\pi \neq \pi' \wedge \texttt{head}(\texttt{fragment}(\widehat{node}_{\pi'})).\text{PRED} = \&\text{INCS}) \Rightarrow$
   $\widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_\pi))$
   $\wedge\ \texttt{head}(\texttt{fragment}(\widehat{node}_\pi)).\text{PRED} = \&\text{EXIT} \Rightarrow (\widehat{PC}_\pi \in [\mathbf{28}, \mathbf{29}] \vee$
   $(\pi \neq \pi' \wedge \texttt{head}(\texttt{fragment}(\widehat{node}_{\pi'})).\text{PRED} = \&\text{EXIT} \wedge \widehat{PC}_{\pi'} \notin [\mathbf{28}, \mathbf{29}]) \Rightarrow$
   $\widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_\pi))$
   $\wedge\ (|\texttt{fragment}(\widehat{node}_\pi)| > 1 \Rightarrow$
   $((\widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_\pi) \wedge \widehat{node}_{\pi'} \neq \texttt{head}(\texttt{fragment}(\widehat{node}_\pi))) \Rightarrow \widehat{PC}_{\pi'} \in \{\mathbf{15}, \mathbf{25}\}))$
8. $\forall \pi \in \Pi, PC_\pi \in \{\mathbf{12}, \mathbf{13}\} \Rightarrow (mynode_\pi \in \mathcal{N}' \wedge (\forall q \in \mathcal{P}, \text{NODE}[q] \neq mynode_\pi \wedge \text{NODE}[q].\text{PRED} \neq mynode_\pi)$
   $\wedge\ mynode_\pi.\text{CS\_SIGNAL} = 0 \wedge mynode_\pi.\text{NONNIL\_SIGNAL} = 0$
   $\wedge\ mynode_\pi = \texttt{head}(\texttt{fragment}(mynode_\pi)) \wedge |\texttt{fragment}(mynode_\pi)| = 1$
   $\wedge\ \texttt{fragment}(mynode_\pi) \neq \texttt{fragment}(\text{TAIL}) \wedge mynode_\pi.\text{PRED} = \text{NIL})$

Figure 6.C.4: (Continued from Figure 6.C.3.) Invariant for the k-ported recoverable mutual exclusion algorithm from Figures 6-3-6-4. (Continued in Figure 6.C.5.)

Lemma 6.C.1 (Mutual Exclusion). *At most one process is in the CS in every configuration of every run.*

Proof. Suppose there are two processes π_i and π_j that are both in CS in a configuration C. There-

Conditions (Continued from Figure 6.C.4):

9. $\forall \pi \in \Pi,\ PC_\pi = \mathbf{14} \Rightarrow (\widehat{node}_\pi \in \mathcal{N}' \wedge \widehat{node}_\pi.\textsc{Pred} = \text{NIL} \wedge \widehat{node}_\pi = \texttt{head}(\texttt{fragment}(\widehat{node}_\pi))$
$\wedge\ \widehat{node}_\pi.\textsc{CS\_Signal} = \texttt{0} \wedge \widehat{node}_\pi.\textsc{NonNil\_Signal} = \texttt{0}$
$\wedge\ (\forall \pi' \in \Pi, (\pi' \neq \pi \wedge \widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_\pi)) \Rightarrow \widehat{PC_{\pi'}} \in \{\mathbf{15}, \mathbf{25}\})$
$\wedge\ mypred_\pi \in \mathcal{N}' \wedge mypred_\pi = \texttt{tail}(\texttt{fragment}(mypred_\pi))$
$\wedge\ (mypred_\pi.\textsc{CS\_Signal} = \texttt{1}$
$\vee\ (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{node}_{\pi'} = mypred_\pi \wedge \widehat{PC_{\pi'}} \in \{\mathbf{14}, \mathbf{15}\} \cup [\mathbf{25}, \mathbf{28}]))$
$\wedge\ (mypred_\pi.\textsc{Pred} = \&\textsc{InCS} \Rightarrow (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} = \mathbf{27} \wedge mypred_\pi = \widehat{node}_{\pi'}))$
$\wedge\ (mypred_\pi.\textsc{Pred} = \&\textsc{Exit} \Rightarrow (((\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} \in [\mathbf{28}, \mathbf{29}]$
$\wedge\ mypred_\pi = \widehat{node}_{\pi'}) \vee (\forall p' \in \mathcal{P}, \textsc{Node}[p'] \neq mypred_\pi)) \wedge |\mathcal{Q}| = 0))$
$\wedge\ (mypred_\pi.\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\} \Rightarrow$
$(\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} \in [\mathbf{14}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{26}] \wedge mypred_\pi = \widehat{node}_{\pi'}))$
$\wedge\ (\texttt{head}(\texttt{fragment}(mypred_\pi)).\textsc{Pred} \in \{\text{NIL}, \&\textsc{Crash}\} \Rightarrow (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} = \mathbf{14}$
$\wedge\ mypred_\pi = \texttt{tail}(\texttt{fragment}(\widehat{node}_{\pi'})) \wedge \widehat{node}_{\pi'} = \texttt{head}(\texttt{fragment}(\widehat{node}_{\pi'}))))$
$\wedge\ \texttt{fragment}(\widehat{node}_\pi) \neq \texttt{fragment}(mypred_\pi))$

10. $\forall \pi \in \Pi, ((\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\} \wedge \widehat{node}_\pi.\textsc{Pred} = \text{NIL}) \Rightarrow (PC_\pi = \widehat{PC_\pi} \vee PC_\pi \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{18}]))$
$\wedge\ ((\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\} \wedge \widehat{node}_\pi.\textsc{Pred} = \&\textsc{Crash}) \Rightarrow PC_\pi \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{21}] \cup [\mathbf{23}, \mathbf{24}] \cup [\mathbf{30}, \mathbf{48}])$

11. $\forall \pi \in \Pi, (PC_\pi \in [\mathbf{19}, \mathbf{21}] \cup [\mathbf{23}, \mathbf{24}] \cup [\mathbf{30}, \mathbf{48}] \wedge \widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\}) \Rightarrow \widehat{node}_\pi.\textsc{Pred} = \&\textsc{Crash}$

12. $\forall \pi \in \Pi, (\widehat{PC_\pi} = \mathbf{13} \Rightarrow |\texttt{fragment}(\widehat{node}_\pi)| = 1)$
$\wedge\ (\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\} \Rightarrow (\widehat{node}_\pi = \texttt{head}(\texttt{fragment}(\widehat{node}_\pi)) \wedge \texttt{fragment}(\widehat{node}_\pi) \neq \texttt{fragment}(\textsc{Tail})))$
$\wedge\ (\widehat{PC_\pi} = \mathbf{14} \Rightarrow (\forall \pi' \in \Pi, (\pi' \neq \pi \wedge \widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_\pi)) \Rightarrow \widehat{PC_{\pi'}} \in \{\mathbf{15}, \mathbf{25}\}))$

13. $\forall \pi \in \Pi,\ PC_\pi \in \{\mathbf{15}, \mathbf{25}\} \Rightarrow (\widehat{node}_\pi \in \mathcal{N}' \wedge \widehat{node}_\pi.\textsc{Pred} = mypred_\pi \wedge mypred_\pi \in \mathcal{N}')$

14. $\forall \pi \in \Pi, \widehat{PC_\pi} \in \{\mathbf{15}, \mathbf{25}\} \Rightarrow (\widehat{node}_\pi \in \mathcal{N}' \wedge \widehat{node}_\pi.\textsc{Pred} \in \mathcal{N}'$
$\wedge\ (\widehat{node}_\pi.\textsc{Pred}.\textsc{CS\_Signal} = \texttt{1}$
$\vee\ (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{node}_{\pi'} = \widehat{node}_\pi.\textsc{Pred} \wedge \widehat{PC_{\pi'}} \in \{\mathbf{14}, \mathbf{15}\} \cup [\mathbf{25}, \mathbf{28}]))$
$\wedge\ (\widehat{node}_\pi.\textsc{Pred}.\textsc{Pred} = \&\textsc{InCS} \Rightarrow$
$(\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} = \mathbf{27} \wedge \widehat{node}_\pi.\textsc{Pred} = \widehat{node}_{\pi'}))$
$\wedge\ (\widehat{node}_\pi.\textsc{Pred}.\textsc{Pred} = \&\textsc{Exit} \Rightarrow (((\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} \in [\mathbf{28}, \mathbf{29}]$
$\wedge\ \widehat{node}_\pi.\textsc{Pred} = \widehat{node}_{\pi'}) \vee (\forall p' \in \mathcal{P}, \textsc{Node}[p'] \neq \widehat{node}_\pi.\textsc{Pred})) \wedge |\mathcal{Q}| = 0))$
$\wedge\ (\widehat{node}_\pi.\textsc{Pred}.\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\} \Rightarrow$
$(\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} \in [\mathbf{14}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{26}] \wedge \widehat{node}_\pi.\textsc{Pred} = \widehat{node}_{\pi'}))$

15. $\forall \pi \in \Pi, (\widehat{PC_\pi} \in \{\mathbf{15}, \mathbf{25}\} \wedge \texttt{head}(\texttt{fragment}(\widehat{node}_\pi)).\textsc{Pred} \in \{\text{NIL}, \&\textsc{Crash}\}) \Rightarrow$
$(\exists \pi' \in \Pi, \pi' \neq \pi \wedge \widehat{PC_{\pi'}} = \mathbf{14} \wedge \widehat{node}_{\pi'} = \texttt{head}(\texttt{fragment}(\widehat{node}_\pi))$
$\wedge\ (\forall \pi'' \in \Pi, (\pi'' \neq \pi' \wedge \widehat{node}_{\pi''} \in \texttt{fragment}(\widehat{node}_\pi)) \Rightarrow$
$(\widehat{PC_{\pi''}} \in \{\mathbf{15}, \mathbf{25}\} \wedge \widehat{node}_{\pi''}.\textsc{CS\_Signal} = \texttt{0})))$

Figure 6.C.5: (Continued from Figure 6.C.4.) Invariant for the k-ported recoverable mutual exclusion algorithm from Figures 6-3-6-4. (Continued in Figure 6.C.6.)

fore, $\widehat{PC_{\pi_i}} = \mathbf{27}$ and $\widehat{PC_{\pi_j}} = \mathbf{27}$ in C. By definition of $\mathcal{Q}$, $\pi_i \in \mathcal{Q}$ and $\pi_j \in \mathcal{Q}$. Therefore, by Condition 19 of the invariant, one of the two processes is not π_1 in the ordering of processes in $\mathcal{Q}$. Without loss of generality, let $\pi_i = \pi_1$ and π_j be a process coming later in the ordering. Therefore, by Condition 19(d)i, $\widehat{PC_{\pi_j}} \in \{\mathbf{15}, \mathbf{25}\}$, a contradiction. □

Lemma 6.C.2 (Starvation Freedom). *If the total number of crashes in the run is finite and a process is in the Try section and does not subsequently crash, it later enters the CS.*

Conditions (Continued from Figure 6.C.5):

16. $\text{TAIL} \in \mathcal{N}' \wedge \text{TAIL} = \texttt{tail}(\texttt{fragment}(\text{TAIL})) \wedge (\exists i \in [0, k-1], \text{TAIL} = \text{NODE}[i] \vee \text{TAIL.PRED} = \&\text{EXIT})$
$\wedge$ ($\text{TAIL.CS\_SIGNAL} = 1$
$\vee (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{node}_{\pi'} = \text{TAIL} \wedge \widehat{PC_{\pi'}} \in \{14, 15\} \cup [25, 28]))$
$\wedge$ ($\text{TAIL.PRED} = \&\text{INCS} \Rightarrow (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} = 27 \wedge \text{TAIL} = \widehat{node}_{\pi'})$)
$\wedge$ ($\text{TAIL.PRED} = \&\text{EXIT} \Rightarrow (((\exists \pi' \in \Pi, \widehat{PC_{\pi'}} \in [28, 29] \wedge \text{TAIL} = \widehat{node}_{\pi'}) \vee (\forall p' \in \mathcal{P}, \text{NODE}[p'] \neq \text{TAIL}))$
$\wedge\ |\mathcal{Q}| = 0))$
$\wedge$ ($\text{TAIL.PRED} \notin \{\&\text{INCS}, \&\text{EXIT}\} \Rightarrow (\exists \pi' \in \Pi, \widehat{PC_{\pi'}} \in [14, 15] \cup [25, 26] \wedge \text{TAIL} = \widehat{node}_{\pi'})$))
$\wedge$ ($\texttt{head}(\texttt{fragment}(\text{TAIL})).\text{PRED} \in \{\text{NIL}, \&\text{CRASH}\} \Rightarrow (\exists \pi' \in \Pi, \widehat{PC_{\pi'}} = 14$
$\wedge\ \text{TAIL} = \texttt{tail}(\texttt{fragment}(\widehat{node}_{\pi'})) \wedge \widehat{node}_{\pi'} = \texttt{head}(\texttt{fragment}(\widehat{node}_{\pi'}))))$
$\wedge$ $((\exists \pi \in \Pi, \widehat{PC_{\pi}} \in [14, 15] \cup [25, 29]) \Leftrightarrow (\exists \pi' \in \Pi, \text{TAIL} = \widehat{node}_{\pi'} \wedge \widehat{PC_{\pi'}} \in [14, 15] \cup [25, 29]))$

17. $\forall \pi \in \Pi, ((PC_{\pi} \in [24, 29] \cup [30, 49] \vee \widehat{PC_{\pi}} \in [25, 29]) \Rightarrow \widehat{node}_{\pi}.\text{NONNIL\_SIGNAL} = 1)$
$\wedge$ $(\widehat{PC_{\pi}} = 29 \Rightarrow \widehat{node}_{\pi}.\text{CS\_SIGNAL} = 1)$

18. $|\mathcal{Q}| = 0 \Rightarrow ((\text{TAIL.PRED} = \&\text{EXIT} \vee \exists \pi \in \Pi, (\widehat{PC_{\pi}} = 14 \wedge \text{TAIL} = \texttt{tail}(\texttt{fragment}(\widehat{node}_{\pi}))$
$\wedge\ \widehat{node}_{\pi} = \texttt{head}(\texttt{fragment}(\widehat{node}_{\pi}))))$
$\wedge\ (\forall \pi' \in \Pi, PC_{\pi'} \in [11, 15] \cup \{25\} \cup [28, 29]))$

19. If $|\mathcal{Q}| = l > 0$, then there is an order $\pi_1, \pi_2, \ldots, \pi_l$ of distinct processes in Q such that:

    (a) $\widehat{PC_{\pi_1}} \in \{15\} \cup [25, 27]$

    (b) $(\exists \pi \in \Pi, \widehat{PC_{\pi}} \in [28, 29] \wedge \widehat{node}_{\pi_1}.\text{PRED} = \widehat{node}_{\pi})$
    $\vee\ (\widehat{node}_{\pi_1}.\text{PRED} \in \mathcal{N}' - \{\widehat{node}_{\pi'} | \pi' \in \Pi \wedge \widehat{node}_{\pi'} \neq \text{NIL}\})$

    (c) $\widehat{PC_{\pi_1}} \in \{15, 25\} \Rightarrow (\widehat{node}_{\pi_1}.\text{PRED.CS\_SIGNAL} = 1 \vee$
    $(\exists \pi' \in \Pi, \pi_1 \neq \pi' \wedge \widehat{node}_{\pi'} = \widehat{node}_{\pi_1}.\text{PRED} \wedge \widehat{PC_{\pi'}} = 28))$

    (d) $\forall i \in [2, l]$:

        i. $\widehat{PC_{\pi_i}} \in \{15, 25\}$

        ii. $\widehat{node}_{\pi_i}.\text{PRED} = \widehat{node}_{\pi_{i-1}}$
        Observation: $\widehat{node}_{\pi_i} \in \texttt{fragment}(\widehat{node}_{\pi_1})$.

    (e) $\widehat{node}_{\pi_l} = \texttt{tail}(\texttt{fragment}(\widehat{node}_{\pi_1}))$

    (f) $\widehat{node}_{\pi_1} = \texttt{head}(\texttt{fragment}(\widehat{node}_{\pi_1})) \vee \widehat{node}_{\pi_1}.\text{PRED.PRED} = \&\text{EXIT}$

    (g) $\forall \pi \in \Pi, \pi \neq \pi_1 \Rightarrow \widehat{PC_{\pi}} \in [11, 15] \cup \{25\} \cup [28, 29]$

    (h) $\forall \pi \in \Pi, (\pi \neq \pi_1 \wedge \widehat{node}_{\pi} \neq \text{NIL} \wedge \widehat{node}_{\pi}.\text{PRED} \in \mathcal{N}') \Rightarrow (\widehat{node}_{\pi}.\text{PRED.CS\_SIGNAL} = 0)$

Observation: $\forall \pi \in \Pi, \pi \neq \pi_1 \Rightarrow \widehat{PC_{\pi}} \neq 27$.
Proof: If $\pi \in \mathcal{Q}$, then by Condition 19(d)i, $\widehat{PC_{\pi}} \neq 27$. If $\pi \notin \mathcal{Q}$, then, $\widehat{PC_{\pi}} \neq 27$, by definition of $\mathcal{Q}$. □

Figure 6.C.6: (Continued from Figure 6.C.5.) Invariant for the k-ported recoverable mutual exclusion algorithm from Figures 6-3-6-4.

Proof. As noted in the statement of the claim, we assume that the total number of crashes in the run is finite.

A process π using a port p would not enter the CS during its passage if PC_{π} is forever stuck at a certain line in the algorithm before entering the CS. Hence, in order to prove starvation freedom we have to argue that PC_{π} advances to the next line for every step in the algorithm. An inspection of the Try section reveals that π has procedure calls at Lines **15**, **23**, and **25**, and inside the CS of RLock at Line **35**. Since we require the RLock to be a recoverable starvation-free mutual

exclusion lock, any process that executes Line **24** is guaranteed to eventually reach Line **30** of the Critical section of RLock (and hence reaches Line **35**). Particularly, Golab and Ramaraju's read-write based recoverable extension of Yang and Anderson's lock (see Section 3.2 in [78]) is one such lock that also guarantees a wait-free exit. Of these procedure calls, only the ones at Lines **25** and **35** concern us in the proof, since their implementation involves a wait loop. Therefore, if all the calls to `wait` are shown to complete, π is guaranteed to enter the CS eventually.

We comment on a few other steps in the algorithm as follows before diving into the proof. The **for** loop at Line **32** executes for k iterations, therefore, Lines **32**-**38** execute a bounded number of times. Computing the set of maximal paths at Line **39** is a local computation step and has a bounded time algorithm, therefore, the step is executed a bounded number of times. The set $Paths_\pi$ is a finite set and finding the path $mypath_\pi$ at Line **40** is a local computation step which has a bounded time algorithm, therefore, the step is executed a bounded number of times. Similarly, Line **41** is a local computation step which has a bounded time algorithm, therefore, the step is executed a bounded number of times. As observed above, $Paths_\pi$ is a finite set, therefore the loop at Line **42** iterates a finite number of times. Hence, Lines **42**-**45** execute a bounded number of times. Note, since our algorithm has a wait-free exit (see Lemma 6.C.3), π goes back to the Remainder section in a bounded number of normal steps once it finishes the CS. From the above it follows that π executes wait loops inside the calls for `wait` only at Lines **25** and **35**. Therefore, we consider these two cases where π could potentially loop as follows and ensure that it eventually gets past these lines.

Case 1: π completes the step at Line **35**.

When $PC_\pi = \mathbf{35}$, by Condition 30, cur_π.NonNil_Signal = `1` or $(\exists \pi' \in \Pi, \pi \neq \pi' \wedge cur_\pi = \widehat{node}_{\pi'} \wedge \widehat{PC_{\pi'}} \in [\mathbf{13}, \mathbf{15}])$. Suppose cur_π.NonNil_Signal = `1`. cur_π.NonNil_Signal is an instance of the Signal object from Section 6.2.1, it follows that the call to cur_π.NonNil_Signal.`wait`() on Line **35** returns in a wait-free manner. Therefore, π completes the step at Line **35**.

Assume cur_π.NonNil_Signal $\neq$ `1` and $(\exists \pi' \in \Pi, \pi \neq \pi' \wedge cur_\pi = \widehat{node}_{\pi'} \wedge \widehat{PC_{\pi'}} \in [\mathbf{13}, \mathbf{15}])$. Suppose $PC_{\pi'} = \widehat{PC_{\pi'}}$ and there are no crash steps by π' before completing Line **15**. In that case π' executes cur_π.NonNil_Signal.`set`() to completion at Line **15** and sets cur_π.NonNil_Signal = `1` in a wait-free manner. It follows that the call to cur_π.NonNil_Signal.`wait`() on Line **35** returns subsequently in a wait-free manner. Therefore, assume that $PC_{\pi'} \neq \widehat{PC_{\pi'}}$. By Conditions 6, 17 and the fact that cur_π.NonNil_Signal $\neq$ `1`, $PC_{\pi'} \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{21}] \cup \{\mathbf{23}\}$. Therefore, π' eventually executes cur_π.NonNil_Signal.`set`() to completion at Line **23** and sets cur_π.NonNil_Signal =

1 in a wait-free manner. It follows that the call to cur_π.NonNil_Signal.wait() on Line 35 returns subsequently in a wait-free manner. Note, in case of a crash by π' before executing cur_π.NonNil_Signal.set() to completion, π' starts at Line 10 and reaches Line 23. This is because $\widehat{PC_{\pi'}} \in [13, 15]$ implies Node$[\widehat{port_{\pi'}}] \neq$ NIL and $\widehat{node}_{\pi'}$.Pred $\notin \{\&\text{InCS}, \&\text{Exit}\}$. Therefore, the **if** conditions at Lines 10, 20, and 21 are not met and π' reaches Line 23. From the above it follows that π completes the step at Line 35. ■

Case 2: π completes the step at Line 25.

In order to argue that π completes the step at Line 25, we consider two cases. For the first case we have head(fragment($\widehat{node}_\pi$)).Pred $\in \{\&\text{InCS}, \&\text{Exit}\}$ and the second occurs when head(fragment($\widehat{node}_\pi$)).Pred $\in \{\text{NIL}, \&\text{Crash}\}$. The first case occurs when $\pi \in \mathcal{Q}$ and the second occurs when $\pi \notin \mathcal{Q}$, both because of the value of head(fragment($\widehat{node}_\pi$)).Pred. We argue both the cases as follows.

Case 2.1: head(fragment($\widehat{node}_\pi$)).Pred $\in \{\&\text{InCS}, \&\text{Exit}\}$.

By definition of $\mathcal{Q}$, $\pi \in \mathcal{Q}$. By Condition 19, there is an ordering $\pi_1, \pi_2, \ldots, \pi_l$ of the processes in $\mathcal{Q}$, and π appears somewhere in that ordering. Assume for a contradiction that there is a run R in which π never completes the step at Line 25. Therefore, in R there are some processes (including π) in $\mathcal{Q}$ that initiate the passage but never enter the CS. Since the processes never enter the CS, after a certain configuration they are forever stuck at Line 25. Let $\pi_j \in \mathcal{Q}$ be the process in R that forever loops at Line 25, such that it has the least index j according to the ordering defined by Condition 19. Let C be the earliest configuration in R such that all the processes appearing before π_j in the ordering defined by Condition 19 have gone back to the Remainder section after completing the CS and π_j is still stuck at Line 25. Since those processes are no more queued processes, π_j appears first in the ordering, i.e., $\pi_j = \pi_1$. Since $PC_\pi = 25$, by Condition 19c, $\widehat{node}_{\pi_j}$.Pred.CS_Signal = 1 or $(\exists \pi' \in \Pi, \pi_j \neq \pi' \wedge \widehat{node}_{\pi'} = \widehat{node}_{\pi_j}.\text{Pred} \wedge \widehat{PC_{\pi'}} = 28)$. If $\widehat{node}_{\pi_j}$.Pred.CS_Signal = 1, then π_j returns from the call to $\widehat{node}_{\pi_j}$.Pred.CS_Signal.wait() at Line 25 completing the step. Otherwise, suppose $\widehat{node}_{\pi_j}$.Pred.CS_Signal $\neq$ 1 and $(\exists \pi' \in \Pi, \pi_j \neq \pi' \wedge \widehat{node}_{\pi'} = \widehat{node}_{\pi_j}.\text{Pred} \wedge \widehat{PC_{\pi'}} = 28)$. If $PC_{\pi'} = \widehat{PC_{\pi'}}$, then π' eventually executes $\widehat{node}_{\pi'}$.CS_Signal.set() to completion at Line 28 and sets $\widehat{node}_{\pi'}$.CS_Signal = 1 in a wait-free manner. It follows that the call

to $\widehat{node}_{\pi_j}$.PRED.CS_SIGNAL.`wait`() at Line **25** returns subsequently in a wait-free manner. If $PC_{\pi'} \neq \widehat{PC_{\pi'}}$, then, by Condition 6, $PC_\pi \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{22}]$. By Condition 1, NODE$[\widehat{port_{\pi'}}] \neq$ NIL and $\widehat{node}_{\pi'}$.PRED $=$ &EXIT. Therefore, the **if** conditions at Lines **10** and **20** are not met, but the one at Line **21** is met and π' executes Line **28** as written in Line **22**. Therefore, π' eventually executes $\widehat{node}_{\pi'}$.CS_SIGNAL.`set`() to completion at Line **28** and sets $\widehat{node}_{\pi'}$.CS_SIGNAL $=$ `1` in a wait-free manner. It follows that the call to $\widehat{node}_{\pi_j}$.PRED.CS_SIGNAL.`wait`() at Line **25** returns subsequently in a wait-free manner. Thus π_j eventually enters the CS by completing the remaining Try section at Line **26**. This contradicts the assumption that π_j is a process in $\mathcal{Q}$ with the least index j defined by the ordering by Condition 19. Therefore, we conclude that π itself completes the step at Line **25** and eventually enters the CS.

Case 2.2: `head`(`fragment`($\widehat{node}_\pi$)).PRED $\in$ {NIL, &CRASH}.

Let C be a configuration when $\widehat{PC_\pi} =$ **25** and `head`(`fragment`($\widehat{node}_\pi$)).PRED $\in$ {NIL, &CRASH}. By Condition 15, $\exists \pi_{i_1} \in \Pi, \pi_{i_1} \neq \pi \wedge \widehat{PC_{\pi_{i_1}}} =$ **14** $\wedge\ \widehat{node}_{\pi_{i_1}} =$ `head`(`fragment`($\widehat{node}_\pi$)). Suppose $\widehat{node}_{\pi_{i_1}}$.PRED $=$ NIL. By Condition 10, $PC_{\pi_{i_1}} = \widehat{PC_{\pi_{i_1}}}$ (or $PC_{\pi_{i_1}} \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{18}]$), we cover this case later). By Condition 13, $mypred_{\pi_{i_1}} \in \mathcal{N}'$. If π_{i_1} takes normal steps at Line **14**, then it sets $\widehat{node}_{\pi_{i_1}}$.PRED $=$ $mypred_{\pi_{i_1}}$ and sets $\widehat{PC_{\pi_{i_1}}} =$ **15**. We hold the argument for the current case when $\widehat{node}_{\pi_{i_1}}$.PRED $=$ NIL briefly and argue the case when $\widehat{node}_{\pi_{i_1}}$.PRED $=$ &CRASH as follows and then join the two arguments (i.e., $\widehat{node}_{\pi_{i_1}}$.PRED $\in$ {NIL, &CRASH}) later. So now assume that $\widehat{node}_{\pi_{i_1}}$.PRED $=$ &CRASH (this covers the case when $\widehat{node}_{\pi_{i_1}}$.PRED $=$ NIL and $PC_{\pi_{i_1}} \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{18}]$), since $\widehat{node}_{\pi_{i_1}}$.PRED $=$ &CRASH at Line **18** eventually). By Condition 10, $PC_{\pi_{i_1}} \in \{\mathbf{10}\} \cup [\mathbf{16}, \mathbf{21}] \cup [\mathbf{23}, \mathbf{24}] \cup [\mathbf{30}, \mathbf{48}]$. For every value of $PC_{\pi_{i_1}}$, it follows that π_{i_1} eventually executes Line **49** (note, by Case 1 above, π_{i_1} completes all steps at Line **35**). Once π_{i_1} executes Line **49**, it sets $\widehat{PC_{\pi_{i_1}}} =$ **25**. Hence, in both cases (i.e., $\widehat{node}_{\pi_{i_1}}$.PRED $\in$ {NIL, &CRASH}) $\widehat{PC_{\pi_{i_1}}} =$ **25** eventually. Let C' be the earliest configuration after C when $\widehat{PC_{\pi_{i_1}}} =$ **25**, by Condition 1, $\widehat{node}_{\pi_{i_1}}$.PRED $\in \mathcal{N}'$ in C'. If `head`(`fragment`($\widehat{node}_\pi$)).PRED $\in$ {&INCS, &EXIT} in C', then by the same argument as in Case 2.1 we are done. Otherwise, again by Condition 15, $\exists \pi_{i_2} \in \Pi, \pi_{i_2} \neq \pi \wedge \widehat{PC_{\pi_{i_2}}} =$ **14** $\wedge\ \widehat{node}_{\pi_{i_2}} =$

$\texttt{head}(\texttt{fragment}(\widehat{node}_\pi))$.

We now show as follows that $\texttt{head}(\texttt{fragment}(\widehat{node}_\pi)).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$ eventually. Assume to the contrary that $\texttt{head}(\texttt{fragment}(\widehat{node}_\pi)).\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\}$ forever. We know there are k active processes, and by Conditions 3, 4, and 7, there are a finite number of distinct fragments. Applying the above argument about π and π_{i_1} inductively on these fragments, the fragments increase in size monotonically and we get to a configuration such that each process satisfies one of three cases as follows: (i) the process has its node appear in $\texttt{fragment}(\widehat{node}_\pi)$, (ii) there is a process π_{i_3} such that $\texttt{head}(\texttt{fragment}(\widehat{node}_{\pi_{i_3}})).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$ and the process has its node appear in $\texttt{fragment}(\widehat{node}_{\pi_{i_3}})$, or (iii) the process is in the Remainder section after completing the super-passage. Let C'' be the earliest such configuration. In C'' we have $\exists \pi_{i_4} \in \Pi, \pi_{i_4} \neq \pi \wedge \widehat{PC_{\pi_{i_4}}} = \mathbf{14} \wedge \widehat{node}_{\pi_{i_4}} = \texttt{head}(\texttt{fragment}(\widehat{node}_\pi))$. We can now apply the above argument about π and π_{i_1} on π and π_{i_4}. We continue to do so until we get to a configuration where each process satisfies one of the following two cases: (i) the process has its node appear in $\texttt{fragment}(\widehat{node}_\pi)$, (ii) the process forever remains in the Remainder section after completing the super-passage. Let C''' be earliest such configuration where we have $\exists \pi_{i_5} \in \Pi, \pi_{i_5} \neq \pi \wedge \widehat{PC_{\pi_{i_5}}} = \mathbf{14} \wedge \widehat{node}_{\pi_{i_5}} = \texttt{head}(\texttt{fragment}(\widehat{node}_\pi))$. Note, we have $\widehat{PC_{\pi_{i_5}}} = \mathbf{14}$ in C''', and every other process π' has either $\widehat{PC_{\pi'}} = \mathbf{11}$ (for being in the Remainder section) or $\widehat{PC_{\pi'}} \in \{\mathbf{15}, \mathbf{25}\}$ (for being in $\texttt{fragment}(\widehat{node}_\pi) = \texttt{fragment}(\widehat{node}_{\pi_{i_5}})$) for all configurations after C'''. We can apply the above argument about π and π_{i_1} on π and π_{i_5} so that $\exists \pi_{i_6} \in \Pi, \pi_{i_6} \neq \pi \wedge \widehat{PC_{\pi_{i_6}}} = \mathbf{14} \wedge \widehat{node}_{\pi_{i_6}} = \texttt{head}(\texttt{fragment}(\widehat{node}_\pi))$. This contradicts the above conclusion that only $\widehat{PC_{\pi_{i_5}}} = \mathbf{14}$ in all configurations after C'''. Therefore we conclude that our assumption that $\texttt{head}(\texttt{fragment}(\widehat{node}_\pi)).\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\}$ forever is incorrect and $\texttt{head}(\texttt{fragment}(\widehat{node}_\pi)).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$ eventually. Hence, by the same argument as in Case 2.1 we are done.

From the above it follows that π completes the step at Line **25**. ■

From the above it follows that π completes the steps at Lines **25** and **35** whenever it encounters them during the passage. Therefore, the algorithm satisfies starvation freedom. □

Lemma 6.C.3 (Wait-free Exit). *There is a bound b such that, if a process π is in the Exit*

section, and executes steps without crashing, π completes the Exit section in at most b of its steps.

Proof. An inspection of the algorithm reveals that Lines **27-29** do not involve repeated execution of any steps. The implementation of the Signal object from Figure 6-2 shows that the code for $\mathcal{X}$.`set`() does not involve a loop, Hence, the call to $mynode_\pi$.CS_Signal.`set`() at Line **28** terminates. Hence the claim. □

Lemma 6.C.4 (Wait-Free CSR). *There is a bound b such that, if a process crashes while in the CS, it reenters the CS before completing b consecutive steps without crashing.*

Proof. Suppose π crashes while in the CS, i.e., when $\widehat{PC_\pi}$ = **27**, π crashes. By Condition 1, Node$[\widehat{port_\pi}] \neq$ NIL and $\widehat{node_\pi}$.Pred = &InCS. Therefore, when π restarts from the crash and starts executing at Line **10**, it finds that the **if** conditions at Lines **10** and **18** are not met. It therefore reaches Line **20** with $mypred_\pi = \widehat{node_\pi}$.Pred (by Condition 2) by executing Lines **10**, **16-19** (none of which are repeatedly executed). The **if** condition at Line **20** is met and π is put into the CS in a wait-free manner. Hence the claim. □

Lemma 6.C.5 (Critical Section Reentry). *If a process π crashes inside the CS, then no other process enters the CS before π reenters the CS.*

Proof. This is immediate from Lemma 6.C.1 and Lemma 6.C.4 as observed in [112]. □

6.D Proof of correctness of Signal object

Proof of Theorem 6.2.1. Let α be the earliest event where some process performed Line **1** and β be the earliest event where some process π' performed Line **8** and does not subsequently fail.

Case 1: α occurs before β.

In this case we linearize the execution as follows:

- every execution of $\mathcal{X}$.`set`() is linearized to its Line **1**,
- every execution of $\mathcal{X}$.`wait`() is linearized to its Line **8**, where $\mathcal{X}$.State is `1` (since α precedes β).

Since α occurs before β, π' notices that Bit = `1` at Line **8** and hence returns from the call to `wait`().

Case 2: β occurs before α.

Consider the execution of $\mathcal{X}$.set() that is the first to complete. Let π be the process that performs this execution of $\mathcal{X}$.set(). At Line 2 π reads $go_{\pi'}$ from GoAddr into $addr_\pi$. Since β occurs before α, $addr_\pi \neq$ NIL, therefore, at Line 4 π sets $*go_{\pi'}$ to *true*. This releases π' from its busy-wait at Line 9. We linearize the call to $\mathcal{X}$.set() by π to its Line 4, and every other complete execution of $\mathcal{X}$.set() in the run to its point of completion. Note, β is the earliest event where some process π' performed Line 8 and does not subsequently fail and we assume that no two executions of $\mathcal{X}$.wait() are concurrent. Therefore, every other execution of $\mathcal{X}$.wait(), happens after the call considered in α sets Bit to 1. This implies that such a call would complete because the calling process would read Bit = 1 at Line 8 and return.

RMR Complexity: It is easy to see that the RMR Complexity of $\mathcal{X}$.set() is $O(1)$ since there are a constant steps in any execution of $\mathcal{X}$.set(). For any execution of $\mathcal{X}$.wait() by a process π, π creates a new boolean at Line 5 that resides in π's memory partition. Therefore, the busy-wait by π at Line 9 incurs a $O(1)$ RMR and the rest of the lines in $\mathcal{X}$.wait() (Lines 5-8) incur a $O(1)$ RMR. □

6.E Proof of invariant

In this section we prove that our algorithm from Figures 6-3-6-4 satisfies the invariant described in Figures 6.C.3-6.C.6. In order to prove that we need support from a few extra conditions that we present in Figures 6.E.1-6.E.3. Therefore, we prove that our algorithm satisfies all the conditions described in Figures 6.C.3-6.E.3.

Lemma 6.E.1. *The algorithm in Figures 6-3-6-4 satisfies the invariant (i.e., the conjunction of the 39 conditions) stated in Figures 6.C.3-6.E.3, i.e., the invariant holds in every configuration of every run of the algorithm.*

Proof. We prove the lemma by induction. Specifically, we show (i) *base case:* the invariant holds in the initial configuration, and (ii) *induction step:* if the invariant holds in a configuration C and a step of a process takes the configuration C to C', then the invariant holds in C'.

In the initial configuration, we have Tail = &SpecialNode, $\forall \pi \in \Pi$, $PC_\pi = 10$, $\widehat{PC_\pi} = 11$, and Node$[\widehat{port_\pi}]$ = NIL. Note, $|\mathcal{Q}| = 0$ by definition of $\mathcal{Q}$. Since all processes are in the Remainder section, Condition 1 holds because of the value of the Node array as noted above. Since $\widehat{node_\pi} =$

Definitions (Continued from Figure 6.C.4):

• $\texttt{owner}(qnode)$ denotes the process that created the $qnode$ at Line 11.

Conditions (Continued from Figure 6.C.6):

20. $\forall \pi \in \Pi, (PC_\pi = 31 \wedge \texttt{head}(\texttt{fragment}(\textsc{Tail})).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}) \Rightarrow$
$\texttt{fragment}(\widehat{node_\pi}) \neq \texttt{fragment}(\textsc{Tail})$
21. $\forall \pi \in \Pi, (PC_\pi \in [32, 41] \Rightarrow tailpath_\pi = \text{NIL}) \wedge (PC_\pi \in [32, 41] \Rightarrow headpath_\pi = \text{NIL})$
$\wedge\ (PC_\pi = 32 \Rightarrow i_\pi \in [0, k]) \wedge (PC_\pi \in [33, 38] \Rightarrow i_\pi \in [0, k-1])$
$\wedge\ (PC_\pi \in [39, 49] \Rightarrow i_\pi = k) \wedge (PC_\pi \in [32, 49] \Rightarrow tail_\pi \in \mathcal{N}')$
22. $\forall \pi \in \Pi, PC_\pi \in [32, 49] \Rightarrow (tail_\pi \in V_\pi \vee (\exists i \in [i_\pi, k-1], tail_\pi = \textsc{Node}[i]) \vee (tail_\pi.\textsc{Pred} = \&\textsc{Exit}))$
23. $\forall \pi \in \Pi$, if $PC_\pi \in [32, 49]$, then:
    (a) (V_π, E_π) is a directed acyclic graph,
    (b) Maximal paths in (V_π, E_π) are disjoint.
24. $\forall \pi \in \Pi$, if $PC_\pi \in [32, 49]$, then one of the following holds (i.e., (a) $\vee$ (b) $\vee$ (c) $\vee$ (d)):
    (a) $\texttt{head}(\texttt{fragment}(tail_\pi)).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$
    (b) there is a unique maximal path σ in the graph (V_π, E_π), such that, $\texttt{end}(\sigma).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$ and $\texttt{start}(\sigma).\textsc{Pred} \neq \&\textsc{Exit}$
    (c) $i_\pi < k$ and $\exists i' \in [i_\pi, k-1], \textsc{Node}[i'].\textsc{Pred}.\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$
    $\wedge\ \texttt{tail}(\texttt{fragment}(\textsc{Node}[i'])).\textsc{Pred} \neq \&\textsc{Exit}$
    (d) $|\mathcal{Q}| = 0$
25. $\forall \pi \in \Pi, (PC_\pi \in [32, 39] \wedge \texttt{head}(\texttt{fragment}(tail_\pi)).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}) \Rightarrow$
$\forall \widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node_\pi}), ((\exists i_\pi < i' < k, \textsc{Node}[i'] = \widehat{node}_{\pi'}) \vee$
$(\widehat{node}_{\pi'} \in V_\pi \wedge (\widehat{node_\pi} \neq \widehat{node}_{\pi'} \Rightarrow (\widehat{node}_{\pi'}, \widehat{node}_{\pi'}.\textsc{Pred}) \in E_\pi)))$
26. $\forall \pi \in \Pi$, If $PC_\pi \in [32, 41]$ and there is a maximal path σ in (V_π, E_π) such that $\texttt{end}(\sigma).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$, then, for an arbitrary vertices v and v' on the path σ,
$\forall \widehat{node} \in \texttt{fragment}(v), ((\exists i_\pi < i' < k, \textsc{Node}[i'] = \widehat{node}) \vee (\widehat{node} \in V_\pi$
$\wedge\ (\widehat{node}.\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\} \Rightarrow (\widehat{node}, \widehat{node}.\textsc{Pred}) \in E_\pi))),$
and $(\texttt{fragment}(v) \neq \texttt{fragment}(v') \Rightarrow$
$(v.\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\} \vee v'.\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}))$

Figure 6.E.1: (Continued from Figure 6.C.6.) Invariant for the k-ported recoverable mutual exclusion algorithm from Figures 6-3-6-4. (Continued in Figure 6.E.2.)

NIL by definition, Condition 4, 7 holds. Since $\textsc{Tail} = \&\textsc{SpecialNode}$, Condition 16 holds. Since $|\mathcal{Q}| = 0$, $\textsc{Tail} = \&\textsc{SpecialNode}$ as noted above, hence, Condition 18 holds. All of the remaining conditions of the invariant hold vacuously in the initial configuration. Hence, we have the base case.

To verify the induction step, Let C be an arbitrary configuration in which the invariant holds, π be an arbitrary process, and C' be the configuration that results when π takes a step from C. In the following, we enumerate each possible step of π and argue that the invariant continues to hold in C', even though the step changes the values of some variables. Since our invariant involves universal quantifiers for all the conditions, we have only argued it thoroughly as it is applicable to π and wherever necessary for another process for the sake of brevity. We also skip arguing about conditions that hold vacuously, are easy to verify, are argued before in a similar way, or need not be

Conditions (Continued from Figure 6.C.6):

27. $\forall \pi \in \Pi$, if $PC_\pi \in [32, 49]$, then:
    (a) $\forall v \in V_\pi, v \in \mathcal{N}' \wedge (i_\pi > \widehat{port_\pi} \Rightarrow mynode_\pi \in V_\pi)$
    (b) $\forall\ 0 \le i' < i_\pi, (\text{NODE}[i'] \in \texttt{fragment}(\widehat{node_\pi}) \wedge \texttt{fragment}(tail_\pi) \neq \texttt{fragment}(\widehat{node_\pi})) \Rightarrow \text{NODE}[i'] \in V_\pi$
    (c) $\forall\ 0 \le i' < i_\pi, \forall v \in V_\pi, (\widehat{port_{\texttt{owner}(v)}} = i' \wedge v \neq \text{NODE}[i']) \Rightarrow (v.\text{PRED} = \&\text{EXIT} \wedge \forall p' \in \mathcal{P}, \text{NODE}[p'] \neq v)$
    (d) $\forall\ 0 \le i' < i_\pi, (\text{NODE}[i'].\text{PRED} \notin \{\&\text{CRASH}, \&\text{INCS}, \&\text{EXIT}\} \wedge \text{NODE}[i'] \in V_\pi) \Rightarrow$
    $(\text{NODE}[i'], \text{NODE}[i'].\text{PRED}) \in E_\pi$
    (e) $\forall\ 0 \le i' < i_\pi, ((\forall v \in V_\pi, \text{NODE}[i'] \neq v) \Rightarrow$
    $((\texttt{head}(\texttt{fragment}(tail_\pi)).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\} \wedge \text{NODE}[i'] \in \texttt{fragment}(tail_\pi))$
    $\vee\ \texttt{head}(\texttt{fragment}(\text{NODE}[i'])).\text{PRED} \in \{\text{NIL}, \&\text{CRASH}\}))$
    (f) $\forall v \in V_\pi, \texttt{end}(v).\text{PRED} = \&\text{CRASH} \Rightarrow \texttt{head}(\texttt{fragment}(v)).\text{PRED} = \&\text{CRASH}$
    (g) $\forall v \in V_\pi$, If there is a pair $(v, u) \in E_\pi$, then $u \in V_\pi$ and $(v.\text{PRED} = u \vee v.\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\})$
    (h) $\forall (u, v) \in E_\pi, (\exists i \in [0, k-1], \text{NODE}[i] = u) \vee (\forall i' \in [0, k-1], \text{NODE}[i'].\text{PRED} \neq v)$
    (i) $\forall (v, w) \in E_\pi, (v.\text{PRED} \in \{w, \&\text{INCS}, \&\text{EXIT}\}) \wedge (v.\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\} \Rightarrow w.\text{PRED} = \&\text{EXIT})$
    (j) $\forall (u, v) \in E_\pi, u \neq mynode_\pi$
    (k) $i_\pi > \widehat{port_\pi} \Rightarrow$ there is a path σ in the graph (V_π, E_π), such that, $\texttt{end}(\sigma) = mynode_\pi$.
28. $\forall \pi \in \Pi, (PC_\pi \in [32, 47] \wedge \texttt{head}(\texttt{fragment}(tail_\pi)).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}) \Rightarrow$
    $\texttt{fragment}(\widehat{node_\pi}) \neq \texttt{fragment}(\text{TAIL})$
    $\wedge\ ((PC_\pi \in [39, 47]) \wedge tail_\pi \notin V_\pi) \Rightarrow \texttt{head}(\texttt{fragment}(tail_\pi)).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\})$
    $\wedge\ ((PC_\pi \in [42, 47]) \wedge tailpath_\pi \neq \text{NIL}) \Leftrightarrow tail_\pi \in V_\pi)$
29. $\forall \pi \in \Pi, (PC_\pi \in [32, 49] \wedge \texttt{head}(\texttt{fragment}(tail_\pi)).\text{PRED} \notin \{\&\text{INCS}, \&\text{EXIT}\}) \Rightarrow$
    $\texttt{head}(\texttt{fragment}(\text{TAIL})).\text{PRED} \notin \{\&\text{INCS}, \&\text{EXIT}\}$
30. $\forall \pi \in \Pi, (PC_\pi = 34 \Rightarrow (cur_\pi = \text{NIL} \vee (cur_\pi \in \mathcal{N}' \wedge (cur_\pi = \text{NODE}[i_\pi] \vee cur_\pi.\text{PRED} = \&\text{EXIT}))))$
    $\wedge\ (PC_\pi = 35 \Rightarrow (cur_\pi.\text{NONNIL\_SIGNAL} = 1 \vee (\exists \pi' \in \Pi, \pi \neq \pi' \wedge cur_\pi = \widehat{node_{\pi'}} \wedge \widehat{PC_{\pi'}} \in [13, 15]))))$
    $\wedge\ (PC_\pi \in [35, 38] \Rightarrow (cur_\pi \in \mathcal{N}' \wedge (cur_\pi = \text{NODE}[i_\pi] \vee cur_\pi.\text{PRED} = \&\text{EXIT})))$
    $\wedge\ (PC_\pi \in [36, 37] \Rightarrow cur_\pi.\text{PRED} \in \{\&\text{CRASH}, \&\text{INCS}, \&\text{EXIT}\} \cup \mathcal{N}')\ \wedge\ (PC_\pi = 38 \Rightarrow curpred_\pi \in \mathcal{N}')$
31. $\forall \pi \in \Pi, ((PC_\pi \in [42, 47] \wedge tailpath_\pi \neq \text{NIL} \wedge \texttt{end}(tailpath_\pi).\text{PRED} \notin \{\&\text{INCS}, \&\text{EXIT}\}) \vee PC_\pi = 48) \Rightarrow$
    $\texttt{head}(\texttt{fragment}(tail_\pi)).\text{PRED} \notin \{\&\text{INCS}, \&\text{EXIT}\}$
32. $\forall \pi \in \Pi$, if $PC_\pi \in [39, 41]$ and $\texttt{head}(\texttt{fragment}(tail_\pi)).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}$, then there is a maximal path σ in the graph (V_π, E_π) such that:
    (a) $\texttt{end}(\sigma) = mynode_\pi$
    (b) $\texttt{start}(\sigma) = \texttt{tail}(\texttt{fragment}(mynode_\pi))$
33. $\forall \pi \in \Pi$, if $PC_\pi \in [41, 49]$, then $mypath_\pi$ is the unique path in $Paths_\pi$ such that $mynode_\pi$ appears in $mypath_\pi$
34. $\forall \pi \in \Pi$, if $PC_\pi = 39$ and for every maximal path σ in (V_π, E_π), $\neg(\texttt{end}(\sigma).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}$
    $\wedge\ \texttt{start}(\sigma).\text{PRED} \neq \&\text{EXIT})$, then $(\texttt{head}(\texttt{fragment}(tail_\pi)).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\} \vee |\mathcal{Q}| = 0)$

Figure 6.E.2: (Continued from Figure 6.E.1.) Invariant for the k-ported recoverable mutual exclusion algorithm from Figures 6-3-6-4. (Continued in Figure 6.E.3.)

argued if the step does not affect the condition. Induction step due to a crash step of π is argued in the end. To aid in reading we have numbered each step according to the value of the program counter wherever possible. For the purpose of this proof we assume that π executes the algorithm using the port p, hence $\widehat{port_\pi} = p$.

10 (a). π executes Line **10** when $\widehat{PC_\pi} \in \{11, 12\}$.

Conditions (Continued from Figure 6.C.6):

35. $\forall \pi \in \Pi$, if $PC_\pi = \mathbf{39}$ and there is a maximal path σ in (V_π, E_π), such that,
$\texttt{end}(\sigma).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\} \wedge \texttt{start}(\sigma).\textsc{Pred} \neq \&\textsc{Exit}$, then
$(\texttt{head}(\texttt{fragment}(tail_\pi)).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$
$\vee\ \exists \widehat{node} \in \mathcal{N}, (\widehat{node} = \texttt{start}(\sigma) \wedge \widehat{node} = \texttt{tail}(\texttt{fragment}(\widehat{node}))$
$\wedge\ \texttt{head}(\texttt{fragment}(\widehat{node})).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$
$\wedge\ \texttt{fragment}(\widehat{node_\pi}) \neq \texttt{fragment}(\widehat{node}))$
36. $\forall \pi \in \Pi, (PC_\pi \in \{\mathbf{44}, \mathbf{45}\} \Rightarrow \texttt{end}(\sigma_\pi).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\})$
$\wedge\ (PC_\pi = \mathbf{45} \Rightarrow (|\sigma_\pi| > 1 \wedge \texttt{start}(\sigma_\pi) = \texttt{tail}(\texttt{fragment}(\texttt{start}(\sigma_\pi)))))$
37. $\forall \pi \in \Pi, (PC_\pi \in [\mathbf{40}, \mathbf{48}] \wedge headpath_\pi = \text{NIL}) \Rightarrow$
$(\texttt{head}(\texttt{fragment}(tail_\pi)).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\} \vee |\mathcal{Q}| = 0$
$\vee$ (There is a maximal path σ in (V_π, E_π), such that,
$\texttt{end}(\sigma).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\} \wedge \texttt{start}(\sigma).\textsc{Pred} \neq \&\textsc{Exit}$, and
$\exists \widehat{node} \in \mathcal{N}, (\widehat{node} = \texttt{start}(\sigma) \wedge \widehat{node} = \texttt{tail}(\texttt{fragment}(\widehat{node}))$
$\wedge\ \texttt{head}(\texttt{fragment}(\widehat{node})).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$
$\wedge\ \texttt{fragment}(\widehat{node_\pi}) \neq \texttt{fragment}(\widehat{node}))))$
38. $\forall \pi \in \Pi, (PC_\pi \in [\mathbf{40}, \mathbf{48}] \wedge headpath_\pi \neq \text{NIL}) \Rightarrow$
$((\exists \sigma \in Paths_\pi, headpath_\pi = \sigma) \wedge \texttt{head}(\texttt{fragment}(tail_\pi)).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$
$\vee\ \exists \widehat{node} \in \mathcal{N}, (\widehat{node} = \texttt{start}(headpath_\pi) \wedge \widehat{node} = \texttt{tail}(\texttt{fragment}(\widehat{node}))$
$\wedge\ (\widehat{node}.\textsc{Pred} = \&\textsc{InCS} \Rightarrow (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} = \mathbf{27} \wedge \widehat{node} = \widehat{node_{\pi'}}))$
$\wedge\ (\widehat{node}.\textsc{Pred} = \&\textsc{Exit} \Rightarrow |\mathcal{Q}| = 0)$
$\wedge\ (\widehat{node}.\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\} \Rightarrow$
$(\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} \in \{\mathbf{15}\} \cup [\mathbf{25}, \mathbf{26}] \wedge \widehat{node} = \widehat{node_{\pi'}}))$
$\wedge\ \texttt{head}(\texttt{fragment}(\widehat{node})).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$
$\wedge\ \texttt{fragment}(\widehat{node_\pi}) \neq \texttt{fragment}(\widehat{node}))$
39. $\forall \pi \in \Pi, PC_\pi = \mathbf{49} \Rightarrow (\widehat{PC_\pi} = \mathbf{14} \wedge mypred_\pi \in \mathcal{N}' \wedge mypred_\pi = \texttt{tail}(\texttt{fragment}(mypred_\pi))$
$\wedge\ (mypred_\pi.\textsc{Pred} = \&\textsc{InCS} \Rightarrow (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} = \mathbf{27} \wedge mypred_\pi = \widehat{node_{\pi'}}))$
$\wedge\ (mypred_\pi.\textsc{Pred} = \&\textsc{Exit} \Rightarrow (((\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} \in [\mathbf{28}, \mathbf{29}]$
$\wedge\ mypred_\pi = \widehat{node_{\pi'}}) \vee (\forall p' \in \mathcal{P}, \textsc{Node}[p'] \neq mypred_\pi)) \wedge |\mathcal{Q}| = 0))$
$\wedge\ (mypred_\pi.\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\} \Rightarrow$
$(\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} \in [\mathbf{14}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{26}] \wedge mypred_\pi = \widehat{node_{\pi'}}))$
$\wedge\ (\texttt{head}(\texttt{fragment}(mypred_\pi)).\textsc{Pred} \in \{\text{NIL}, \&\textsc{Crash}\} \Rightarrow (\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{PC_{\pi'}} = \mathbf{14}$
$\wedge\ mypred_\pi = \texttt{tail}(\texttt{fragment}(\widehat{node_{\pi'}})) \wedge \widehat{node_{\pi'}} = \texttt{head}(\texttt{fragment}(\widehat{node_{\pi'}}))))$
$\wedge\ \texttt{fragment}(\widehat{node_\pi}) \neq \texttt{fragment}(mypred_\pi))$

Figure 6.E.3: (Continued from Figure 6.E.2.) Invariant for the k-ported recoverable mutual exclusion algorithm from Figures 6-3-6-4.

In C, $PC_\pi = \mathbf{10}$ and $\widehat{PC_\pi} \in \{\mathbf{11}, \mathbf{12}\}$. By Condition 1, $\textsc{Node}[\widehat{port_\pi}] = \text{NIL}$.

The **if** condition at Line **10** evaluates to *true*. Therefore, this step changes PC_π and $\widehat{PC_\pi}$ to **11**.

Condition 1: As argued above, $\widehat{PC_\pi} = \mathbf{11}$ and $\textsc{Node}[\widehat{port_\pi}] = \text{NIL}$ in C'. Therefore the condition holds in C'.

10 (b). π executes Line **10** when $\widehat{PC_\pi} \notin \{\mathbf{11}, \mathbf{12}\}$.

In C, $PC_\pi = \mathbf{10}$ and $\widehat{PC_\pi} \notin \{\mathbf{11}, \mathbf{12}\}$. By Condition 1, $\textsc{Node}[\widehat{port_\pi}] \neq \text{NIL}$.

The **if** condition at Line **10** evaluates to *false*. Therefore, this step changes PC_π to **16**.

The step does not affect any condition, so the invariant continues to hold in C'.

11. π executes Line **11**.

In C, $PC_\pi = \mathbf{11}$. By Condition 6, $\widehat{PC_\pi} \in [\mathbf{11}, \mathbf{12}]$ in C. By definition, $\widehat{node}_\pi = \text{NIL}$ in C. This step creates a new QNode in the shared memory which gets included in the set $\mathcal{N}$. The node gets a unique address and the variable $mynode_\pi$ holds the address of this new node. This step also initializes this new node so that $mynode_\pi.\text{PRED} = \text{NIL}$, $mynode_\pi.\text{NONNIL\_SIGNAL} = \texttt{0}$, and $mynode_\pi\text{CS\_SIGNAL} = \texttt{0}$. The step then changes PC_π and $\widehat{PC_\pi}$ to **12**.

Condition 4: As argued above, the step creates a new QNode in shared memory that $\widehat{node}_\pi$ is pointing to in C'. The step also initializes $\widehat{node}_\pi.\text{PRED}$ to NIL. Therefore, the condition holds in C'.

Condition 8: Since the step creates a new node in shared memory, no process has a reference to the node except for π in C'. Therefore, $\forall q \in \mathcal{P}, \text{NODE}[q] \neq mynode_\pi \wedge \text{NODE}[q].\text{PRED} \neq mynode_\pi$. Since the nothing except $mynode_\pi$ has a reference to the new node, $mynode_\pi = \texttt{head}(\texttt{fragment}(mynode_\pi))$, $|\texttt{fragment}(mynode_\pi)| = 1$, and $\texttt{fragment}(mynode_\pi) \neq \texttt{fragment}(\text{TAIL})$ holds. Also, as observed above, $mynode_\pi.\text{NONNIL\_SIGNAL} = \texttt{0}$, and $mynode_\pi\text{CS\_SIGNAL} = \texttt{0}$ in C'. Therefore, the condition holds in C'.

12. π executes Line **12**.

In C, $PC_\pi = \mathbf{12}$ and $\widehat{PC_\pi} = \mathbf{12}$. By Condition 8, $mynode_\pi \in \mathcal{N}'$, $\forall q \in \mathcal{P}, \text{NODE}[q] \neq mynode_\pi \wedge \text{NODE}[q].\text{PRED} \neq mynode_\pi$, $mynode_\pi.\text{PRED} = \text{NIL}$, $\texttt{fragment}(mynode_\pi) \neq \texttt{fragment}(\text{TAIL})$, $mynode_\pi = \texttt{head}(\texttt{fragment}(mynode_\pi))$, and $|\texttt{fragment}(mynode_\pi)| = 1$. This step sets $\text{NODE}[\widehat{port_\pi}]$ to $mynode_\pi$ and updates PC_π and $\widehat{PC_\pi}$ to **13**.

Condition 3: As argued above, $mynode_\pi \in \mathcal{N}'$, $mynode_\pi.\text{PRED} = \text{NIL}$, and $|\texttt{fragment}(mynode_\pi)| = 1$ in C'. The step sets $\text{NODE}[\widehat{port_\pi}]$ to $mynode_\pi$, it follows from the above that only $\text{NODE}[\widehat{port_\pi}] = mynode_\pi$ and $\forall q \in \mathcal{P}, q \neq \widehat{port_\pi} \Rightarrow \text{NODE}[q] \neq mynode_\pi$ in C'. Therefore, the condition holds in C'.

Condition 4: By the same argument as for Condition 3 above, we have $\forall q \in \mathcal{P}, q \neq \widehat{port_\pi} \Rightarrow \text{NODE}[q] \neq mynode_\pi$ in C'. Also, $mynode_\pi.\text{PRED} = \text{NIL}$, therefore, from the definition of $\widehat{node}_\pi$ it follows that the condition holds in C'.

Condition 12: As discussed above,$|\texttt{fragment}(mynode_\pi)| = 1$, $\texttt{fragment}(mynode_\pi) \neq \texttt{fragment}(\text{TAIL})$, and $mynode_\pi = \texttt{head}(\texttt{fragment}(mynode_\pi))$ in C, which continues to hold in C'. Therefore, the condition holds in C'.

13. π executes Line **13**.

In C, $PC_\pi = \mathbf{13}$ and $\widehat{PC_\pi} = \mathbf{13}$.

This step performs a FAS operation on the TAIL pointer so that TAIL now points to the same node as pointed by $mynode_\pi$, and sets $mypred_\pi$ to the value held by TAIL in C. It updates PC_π and $\widehat{PC_\pi}$ to **14**.

Condition 9: Applying Condition 16 to C we note that what holds true for TAIL in C, holds true for $mypred_\pi$ in C'. Also applying Condition 8 to C we note that $mynode_\pi \in \mathcal{N}'$, $mynode_\pi$.PRED = NIL, $mynode_\pi = \texttt{head}(\texttt{fragment}(mynode_\pi))$, $|\texttt{fragment}(mynode_\pi)| = 1$, $\texttt{fragment}(mynode_\pi) \neq \texttt{fragment}(\text{TAIL})$, $mynode_\pi$.CS_SIGNAL = 0, and $mynode_\pi$.NONNIL_SIGNAL = 0 in C. In C' it holds that $\texttt{fragment}(mynode_\pi) \neq \texttt{fragment}(mypred_\pi)$. It follows that the condition holds in C'.

Condition 12: The truth value of the condition follows from the reasoning similar to Condition 9 as argued above.

Condition 16: By Condition 8, $mynode_\pi \in \mathcal{N}'$, $|\texttt{fragment}(mynode_\pi)| = 1$, and $mynode_\pi$.PRED = NIL in C. It follows from the same condition that $\texttt{tail}(\texttt{fragment}(mynode_\pi)) = mynode_\pi$. Since the step sets TAIL $= mynode_\pi$ in C' and $\widehat{PC_\pi} = \mathbf{14}$ in C', it follows that Condition 16 holds in C'.

Condition 18: Suppose $|\mathcal{Q}| = 0$ in C. By the condition, $\forall \pi' \in \Pi, PC_{\pi'} \in [\mathbf{11}, \mathbf{15}] \cup \{\mathbf{25}\} \cup [\mathbf{28}, \mathbf{29}]$ in C, which continues to hold in C'. As argued above, $mynode_\pi = \texttt{head}(\texttt{fragment}(mynode_\pi))$ and $|\texttt{fragment}(mynode_\pi)| = 1$, therefore, $mynode_\pi = \texttt{tail}(\texttt{fragment}(mynode_\pi))$ in C and C'. Since TAIL $= mynode_\pi$ in C', the condition holds in C'.

14. π executes Line **14**.

In C, $PC_\pi = \mathbf{14}$ and $\widehat{PC_\pi} = \mathbf{14}$.

The step sets $mynode_\pi$.PRED $= mypred_\pi$ and updates PC_π and $\widehat{PC_\pi}$ to **15**.

Condition 1: By Condition 9, $mypred_\pi \in \mathcal{N}'$ in C. Since the step sets $mynode_\pi$.PRED $= mypred_\pi$, NODE$[\widehat{port_\pi}]$.PRED $\in \mathcal{N}'$ in C'. Therefore, the condition holds in C'.

Condition 3: By Condition 9, $mypred_\pi = \texttt{tail}(\texttt{fragment}(mypred_\pi))$. Therefore, $\forall q \in \mathcal{P}, \text{NODE}[q].\text{PRED} \neq mypred_\pi$ in C. It follows that $\forall q \in \mathcal{P}, \widehat{port_\pi} \neq q \Rightarrow \text{NODE}[q].\text{PRED} \neq mypred_\pi$ in C'. Again from Condition 9 we observe the following about $mypred_\pi$. Either $mypred_\pi$.PRED $=$ &EXIT or $\exists \pi' \in \Pi, \pi \neq \pi' \wedge mypred_\pi = \widehat{node_{\pi'}} \wedge \widehat{PC_{\pi'}} \in [\mathbf{14}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{27}]$ (i.e., NODE$[\widehat{port_\pi}]$.PRED $=$ NODE$[\widehat{port_{\pi'}}]$). Thus the condition holds in C'.

Condition 4: By Condition 4, $|\texttt{fragment}(\widehat{node_\pi})| = b_1 \leq k$ and $|\texttt{fragment}(\widehat{node_\pi})| = b_2 \leq k$

in C. By Condition 7, for a process π', it can not be the case that $\widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_{\pi})$ and $\widehat{node}_{\pi'} \in \texttt{fragment}(mypred_{\pi})$ in C. Therefore, $b_1 + b_2 \leq k$ in C. It follows that in C' $|\texttt{fragment}(\widehat{node}_{\pi})| = b_1 + b_2 \leq k$. Therefore, the condition holds in C'.

Condition 7: $\texttt{fragment}(mypred_{\pi}) = \texttt{fragment}(\widehat{node}_{\pi})$ in C'. Therefore, the condition holds in C' as it held and applied to $\texttt{fragment}(mypred_{\pi})$ in C.

Condition 9: Applying the condition to $\texttt{fragment}(\widehat{node}_{\pi})$ in C, we get that $\forall \pi' \in \Pi, \pi' \neq \pi \wedge \widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_{\pi}) \Rightarrow \widehat{PC}_{\pi'} \in \{\mathbf{15}, \mathbf{25}\}$, which holds in C'. We have $\widehat{PC}_{\pi} = \mathbf{15}$ in C'. Suppose there is a $\pi'' \in \Pi, \pi'' \neq \pi$ such that $\texttt{head}(\texttt{fragment}(mypred_{\pi})) = \widehat{node}_{\pi''}$ and $PC_{\pi''} = \mathbf{14}$ in C. It follows that $\forall \pi' \in \Pi, \pi' \neq \pi'' \wedge \widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_{\pi''}) \Rightarrow \widehat{PC}_{\pi'} \in \{\mathbf{15}, \mathbf{25}\}$ in C'. Therefore, the condition holds for π'' and vacuously for other processes in C'.

Condition 13: Since $\widehat{node}_{\pi}.\textsc{Pred} = mypred_{\pi}$ in C' and invoking the Condition 9 on C and $mypred_{\pi}$, it follows that the condition holds in C'.

Condition 14: This condition holds by an argument similar to Condition 13 as argued above.

Condition 15: Suppose $\texttt{head}(\texttt{fragment}(mypred_{\pi})).\textsc{Pred} \in \{\text{NIL}, \&\textsc{Crash}\}$ in C. It follows that $\exists \pi' \in \Pi, \pi' \neq \pi \wedge \widehat{PC}_{\pi'} = \mathbf{14} \wedge \widehat{node}_{\pi'} = \texttt{head}(\texttt{fragment}(\widehat{node}_{\pi})) \wedge (\forall \pi'' \in \Pi, (\pi'' \neq \pi' \wedge \widehat{node}_{\pi''} \in \texttt{fragment}(\widehat{node}_{\pi})) \Rightarrow \widehat{PC}_{\pi''} \in \{\mathbf{15}, \mathbf{25}\})$ in C'. Therefore, the condition holds in C'.

Condition 19: If $\texttt{head}(\texttt{fragment}(mypred_{\pi})).\textsc{Pred} \in \{\text{NIL}, \&\textsc{Crash}\}$ in C, it follows that $\pi \notin \mathcal{Q}$ in C'. Therefore, it is easy to see that the condition holds in C'. If $\texttt{head}(\texttt{fragment}(mypred_{\pi})).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$ in C, it follows that $\pi \in \mathcal{Q}$ in C'. We have two cases to consider, $|\mathcal{Q}| > 0$ or $|\mathcal{Q}| = 0$ in C. If $|\mathcal{Q}| > 0$ in C, Conditions 19d and 19e are the only ones affected. It is easy to see from the definition of a fragment that these conditions continue to hold in C', therefore, the whole condition would hold in C'. Suppose $|\mathcal{Q}| = 0$ and $mypred_{\pi}.\textsc{Pred} = \&\textsc{Exit}$ in C. It follows that $\pi = \pi_1$ according to the ordering defined by the condition. Condition 19a holds in C' since $\widehat{PC}_{\pi} = \mathbf{15}$ in C'. Condition 19b holds in C' since $mypred_{\pi}.\textsc{Pred} = \&\textsc{Exit}$ in C and C'. Applying Condition 14 to $mypred_{\pi}$, and by the fact that $mypred_{\pi}.\textsc{Pred} = \&\textsc{Exit}$, it follows that Condition 19c holds in C'. Conditions 19d, 19e, and 19f hold in C' by the definition of $\texttt{fragment}(mynode_{\pi})$ and the fact that $mypred_{\pi}.\textsc{Pred} = \&\textsc{Exit}$. Since $|\mathcal{Q}| = 0$ in C, invoking Condition 18 on C we see that Condition 19g holds in C'. Lastly, $\forall \pi' \in \Pi, \pi' \neq \pi \wedge \widehat{node}_{\pi'} \neq \text{NIL} \wedge \widehat{node}_{\pi'}.\textsc{Pred} \in \mathcal{N}'$ we have $\widehat{PC}_{\pi'} \in \{\mathbf{15}, \mathbf{25}\}$ from Condition 1. Since $|\mathcal{Q}| = 0$ in C, $\texttt{head}(\texttt{fragment}(\widehat{node}_{\pi'})).\textsc{Pred} \in \{\text{NIL}, \&\textsc{Crash}\}$ by definition of $\mathcal{Q}$. Therefore, by Condition 15, $\widehat{node}_{\pi'}.\text{CS\_}\textsc{Signal} = 0$ in C, which continues to hold in

C'.

15. π executes Line **15**.

In C, $PC_\pi = \mathbf{15}$ and $\widehat{PC_\pi} = \mathbf{15}$.

The step executes $mynode_\pi$.NonNil_Signal.set() so that $mynode_\pi$.NonNil_Signal = 1 as a result of the step. It also updates PC_π and $\widehat{PC_\pi}$ to **25**.

Condition 17: As discussed above, $mynode_\pi$.NonNil_Signal = 1 and $\widehat{PC_\pi} = \mathbf{25}$ in C'. Therefore, the condition holds in C'.

16. π executes Line **16**.

In C $PC_\pi = \mathbf{16}$ and $\widehat{PC_\pi} \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{29}]$.

This step changes PC_π to **17**.

The step does not affect any condition, so the invariant continues to hold in C'.

17. π executes Line **17**.

In C $PC_\pi = \mathbf{16}$ and $\widehat{PC_\pi} \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{29}]$.

This step sets $mynode_\pi$ to Node$[\widehat{port_\pi}]$ and changes PC_π to **17**.

Condition 2: Since $mynode_\pi = $ Node$[\widehat{port_\pi}]$ in C', the condition holds in C'.

18 (a). π executes Line **17** when $\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\}$.

In C, $PC_\pi = \mathbf{18}$ and $\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\}$. By Condition 1, $\widehat{node_\pi}$.Pred $\in \{$NIL, &Crash$\}$.

This step checks if $\widehat{node_\pi}$.Pred = NIL and sets it to &Crash, if so. It then changes PC_π to **19**.

Condition 1: By the step, $\widehat{node_\pi}$.Pred = &Crash. Therefore, the condition holds in C'.

Condition 10: Since $\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\}$, $\widehat{node_\pi}$.Pred = &Crash, and $PC_\pi = \mathbf{19}$, the condition holds in C'.

Condition 11: The condition holds by the same argument as for Condition 10 above.

18 (b). π executes Line **17** when $\widehat{PC_\pi} \notin \{\mathbf{13}, \mathbf{14}\}$.

In C, $PC_\pi = \mathbf{18}$ and $\widehat{PC_\pi} \notin \{\mathbf{13}, \mathbf{14}\}$. By Condition 1, $\widehat{node_\pi}$.Pred $\notin \{$NIL, &Crash$\}$.

The **if** condition at Line **18** is not met, hence the step changes PC_π to **19**.

The step does not affect any condition, so the invariant continues to hold in C'.

19. π executes Line **19**.

In C $PC_\pi = \mathbf{19}$.

This step sets $mypred_\pi$ to $\text{NODE}[\widehat{port_\pi}].\text{PRED}$ and changes PC_π to **20**.

Condition 2: Since $mypred_\pi = \text{NODE}[\widehat{port_\pi}].\text{PRED}$ in C', the condition holds in C'.

20 (a). π executes Line **20** when $\widehat{PC_\pi} = \mathbf{27}$.

In C $PC_\pi = \mathbf{20}$ and $\widehat{PC_\pi} = \mathbf{27}$. By Condition 1, $\widehat{node_\pi}.\text{PRED} = \&\text{INCS}$. By Condition 2, $mypred_\pi = \widehat{node_\pi}.\text{PRED}$ in C.

In this step the **if** condition is met, therefore, π moves to the CS and updates PC_π to **27**.

The step does not affect any condition, so the invariant continues to hold in C'.

20 (b). π executes Line **20** when $\widehat{PC_\pi} \neq \mathbf{27}$.

In C $PC_\pi = \mathbf{20}$ and $\widehat{PC_\pi} \neq \mathbf{27}$. By Condition 1, $\widehat{node_\pi}.\text{PRED} \neq \&\text{INCS}$. By Condition 2, $mypred_\pi = \widehat{node_\pi}.\text{PRED}$ in C.

In this step the **if** condition is not met, therefore, π updates PC_π to **21**.

The step does not affect any condition, so the invariant continues to hold in C'.

21 (a). π executes Line **21** when $\widehat{PC_\pi} \in \{\mathbf{28}, \mathbf{29}\}$.

In C $PC_\pi = \mathbf{21}$ and $\widehat{PC_\pi} \in \{\mathbf{28}, \mathbf{29}\}$. By Condition 1, $\widehat{node_\pi}.\text{PRED} = \&\text{EXIT}$. By Condition 2, $mypred_\pi = \widehat{node_\pi}.\text{PRED}$ in C.

In this step the **if** condition is met, therefore, π updates PC_π to **22**.

The step does not affect any condition, so the invariant continues to hold in C'.

21 (b). π executes Line **21** when $\widehat{PC_\pi} \notin \{\mathbf{28}, \mathbf{29}\}$.

In C $PC_\pi = \mathbf{21}$ and $\widehat{PC_\pi} \notin \{\mathbf{28}, \mathbf{29}\}$. By Condition 1, $\widehat{node_\pi}.\text{PRED} \neq \&\text{EXIT}$. By Condition 2, $mypred_\pi = \widehat{node_\pi}.\text{PRED}$ in C.

In this step the **if** condition is not met, therefore, π updates PC_π to **23**.

The step does not affect any condition, so the invariant continues to hold in C'.

22. π executes Line **22**.

In C $PC_\pi = \mathbf{22}$ and $\widehat{PC_\pi} \in \{\mathbf{28}, \mathbf{29}\}$.

π sets $\widehat{PC_\pi} = \mathbf{28}$, then executes Lines **28** and **29** as part of the Try section, and then changes PC_π to **10** and $\widehat{PC_\pi}$ to **11**.

For the correctness of the invariant, refer to the induction steps for Lines **28** and **29**, since the execution of the step is same as executing the two lines and then executing a"**go to** Line **10**".

23. π executes Line **23**.

In C $PC_\pi = \mathbf{23}$. By Condition 6, $\widehat{PC_\pi} \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{26}]$.

The step executes $mynode_\pi$.NonNil_Signal.set() so that $mynode_\pi$.NonNil_Signal = 1 as a result of the step. It also updates PC_π to **24**.

Condition 17: As discussed above, $mynode_\pi$.NonNil_Signal = 1 and PC_π = **24** in C'. Therefore, the condition holds in C'.

24. π executes Line **24**.

In C PC_π = **24**. By Condition 6, $\widehat{PC_\pi} \in [\mathbf{13}, \mathbf{15}] \cup [\mathbf{25}, \mathbf{26}]$ in C.

The step executes the Try section of RLock in order to access the Critical Section of RLock starting at Line **30**. Since the RLock is assumed to be satisfying Starvation Freedom, π reaches Line **30** eventually. Hence, the step changes PC_π to **30**. Note, we can use Golab and Ramaraju's [78] read-write based recoverable extension of Yang and Anderson's lock (see Section 3.2 in [78]) as RLock for this purpose.

The step does not affect any condition, so the invariant continues to hold in C'.

25. π executes Line **25**

In C, PC_π = **25** and $\widehat{PC_\pi}$ = **25**. We have $\widehat{node}_\pi \neq$ NIL and $\widehat{node}_\pi$.Pred $\in \mathcal{N}'$ by Condition 1 in C. By Condition 14, either $\widehat{node}_\pi$.Pred.CS_Signal = 1, or $\exists \pi' \in \Pi, \pi \neq \pi' \wedge \widehat{node}_{\pi'} = \widehat{node}_\pi$.Pred $\wedge\ \widehat{PC_{\pi'}} \in \{\mathbf{14}, \mathbf{15}\} \cup [\mathbf{25}, \mathbf{28}]$ in C.

The step executes $mypred_\pi$.CS_Signal.wait() so that the procedure call returns when $mypred_\pi$.CS_Signal = 1. The step also updates PC_π and $\widehat{PC_\pi}$ to **26** when it returns from the procedure call.

Condition 19: Suppose $\widehat{node}_\pi$.Pred.CS_Signal = 1 in C. By Condition 19h, $\pi = \pi_1$ in C. It follows that the condition continues to hold in C' as it held in C. Therefore, assume $\widehat{node}_\pi$.Pred.CS_Signal $\neq$ 1 in C. Since $mypred_\pi$.CS_Signal.wait() returns and the step completes, by the specification of the Signal object, $mypred_\pi$.CS_Signal = 1 in C'. By Condition 5, $mypred_\pi$.Pred = &Exit in C'. Therefore, head(fragment($\widehat{node}_\pi$)).Pred = &Exit in C'. It follows that $\pi \in \mathcal{Q}$ in C' by the definition of $\mathcal{Q}$. By Condition 7, $\forall \pi' \in \Pi, (\pi \neq \pi' \wedge$ head(fragment($\widehat{node}_{\pi'}$)).Pred = &Exit$\wedge \widehat{PC_{\pi'}} \notin [\mathbf{28}, \mathbf{29}]) \Rightarrow \widehat{node}_{\pi'} \in$ fragment($\widehat{node}_\pi$). We now proceed to prove that Condition 19 holds in C' as follows. We have $mypred_\pi$.Pred = &Exit and $mypred_\pi$.CS_Signal = 1 in C'. Therefore, $\forall \pi' \in \Pi, \widehat{node}_{\pi'} = mypred_\pi \Rightarrow \widehat{PC_{\pi'}}$ = **29** in C' by Condition 5. Since $\widehat{node}_\pi$.Pred.Pred = &Exit, Condition 19f holds and it follows that $\pi = \pi_1$. Since $\widehat{PC_\pi}$ = **26**, Condition 19a holds in C'. We also note from the above that Conditions 19b and 19c hold in C'. By the definition of fragment($\widehat{node}_\pi$) and

Condition 7 it follows that Conditions 19d and 19e hold. For any process π', if $\widehat{node_{\pi'}}$.Pred = &Exit, then by Condition 1, $\widehat{PC_{\pi'}} \in \{\mathbf{28}, \mathbf{29}\}$. If $\widehat{node_{\pi'}} \neq \texttt{head}(\texttt{fragment}(\widehat{node_{\pi'}}))$, then by Condition 7, $\widehat{PC_{\pi'}} \in \{\mathbf{15}, \mathbf{25}\}$. If $\widehat{node_{\pi'}}$.Pred $\in \{\text{NIL}, \&\text{Crash}\}$, then by Condition 1, $\widehat{PC_{\pi'}} \in \{\mathbf{13}, \mathbf{14}\}$. By Condition 7 there is only one fragment whose head node has its Pred pointer set to &Exit. It follows that Condition 19g holds from the above. From Conditions 5, 9, 14, and 15 it follows that Condition 19h holds in C'. Therefore, the entire condition holds in C'.

26. π executes Line **26**.

In C $PC_\pi = \mathbf{26}$.

The step sets $\widehat{node_\pi}$.Pred = &InCS, updates PC_π and $\widehat{PC_\pi}$ to **27**, and goes to the CS.

Condition 1: As argued above, $\widehat{node_\pi}$.Pred = &InCS and $\widehat{PC_\pi} = \mathbf{27}$ in C'. Therefore, the condition holds in C'.

Condition 9: Suppose there is a $\pi'' \in \Pi, \pi'' \neq \pi$ such that $mypred_{\pi''} = \widehat{node_\pi}$ in C. It follows that $\widehat{node_\pi}$.Pred = &InCS and $\widehat{PC_\pi} = \mathbf{27}$ in C'. Therefore, the condition holds for π'' and vacuously for other processes in C'.

27. π executes Line **27**.

In C $PC_\pi = \mathbf{27}$.

The step sets $\widehat{node_\pi}$.Pred = &Exit, and updates PC_π and $\widehat{PC_\pi}$ to **28**.

Condition 1: As argued above, $\widehat{node_\pi}$.Pred = &Exit and $\widehat{PC_\pi} = \mathbf{28}$ in C'. Therefore, the condition holds in C'.

Condition 9: Suppose there is a $\pi'' \in \Pi, \pi'' \neq \pi$ such that $mypred_{\pi''} = \widehat{node_\pi}$ in C. It follows that $\widehat{node_\pi}$.Pred = &Exit and $\widehat{PC_\pi} = \mathbf{28}$ in C'. Therefore, the condition holds for π'' and vacuously for other processes in C'.

Condition 18: If $|\mathcal{Q}| > 1$ in C, the condition holds vacuously in C'. If Tail $\neq \widehat{node_\pi}$, then by Condition 7 and 16, the condition holds in C'. Otherwise, suppose $|\mathcal{Q}| = 1$ and Tail $= \widehat{node_\pi}$ in C. Tail.Pred = &Exit in C' by the step. Therefore, the condition holds in C'.

Condition 19: Applying the condition to π in C, $\pi = \pi_1$ according to the ordering of the condition. If $|\mathcal{Q}| = 1$, the condition holds vacuously in C'. Therefore, suppose $|\mathcal{Q}| > 1$ in C. There is a process $\pi' \in \Pi$ such that $\pi' = \pi_2$ according to the ordering and $mypred_{\pi'} = \widehat{node_\pi}$ in C. By Condition 19d, $\widehat{PC_{\pi'}} \in \{\mathbf{15}, \mathbf{25}\}$ in C which continues to hold in C'. Therefore, it follows that Conditions 19a, 19b, and 19c hold for π' in C'. It is easy to see that the rest of

the sub-conditions hold in C' as a result of the step. Therefore, the condition holds in C'.

28. π executes Line **28**.

In C $PC_\pi = \mathbf{28}$ and $\widehat{PC_\pi} = \mathbf{28}$.

The step executes $mynode_\pi$.CS_SIGNAL.`set`() so that $mynode_\pi$.CS_SIGNAL = `1` as a result of the step. It also updates PC_π and $\widehat{PC_\pi}$ to **29**.

Condition 17: As discussed above, $mynode_\pi$.CS_SIGNAL = `1` and $\widehat{PC_\pi} = \mathbf{29}$ in C'. Therefore, the condition holds in C'.

29. π executes Line **29**.

In C $PC_\pi = \mathbf{29}$ and $\widehat{PC_\pi} = \mathbf{29}$.

The step sets NODE$[\widehat{port_\pi}]$ to NIL, sets PC_π to **10**, and $\widehat{PC_\pi}$ to **11**.

Condition 1: As argued above, NODE$[\widehat{port_\pi}]$ = NIL and $\widehat{PC_\pi} = \mathbf{11}$ in C'. Therefore, the condition holds in C'.

Condition 5: By Condition 3 implies that $\forall \pi' \in \Pi, \pi' \neq \pi \Rightarrow \widehat{node_{\pi'}} \neq \widehat{node_\pi}$ in C. By Condition 1, $\widehat{node_\pi}$.PRED = &EXIT in C which holds in C'. By Condition 17, $\widehat{node_\pi}$.CS_SIGNAL = `1` and $\widehat{node_\pi}$.NONNIL_SIGNAL = `1` in C, which holds in C'. Since $\widehat{node_\pi}$ = NIL in C', it follows that for the QNode pointed to by $\widehat{node_\pi}$ in C the condition holds in C'.

Condition 9: Suppose there is a $\pi'' \in \Pi, \pi'' \neq \pi$ such that $mypred_\pi = \widehat{node_{\pi''}}$ and $PC_{\pi''} = \mathbf{14}$ in C. It follows that $\forall p' \in \mathcal{P}$, NODE$[p'] \neq mypred_\pi$ in C'. Therefore, the condition holds for π'' and vacuously for other processes in C'.

30 (a). π executes Line **30** when $\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\}$.

In C, $PC_\pi = \mathbf{30}$ and $\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\}$. By Condition 10, $\widehat{node_\pi}$.PRED = &CRASH. By Condition 2, $mypred_\pi = \widehat{node_\pi}$.PRED in C.

The **if** condition at Line **30** is not met since $\widehat{node_\pi}$.PRED = &CRASH, therefore, PC_π changes to **31**.

Condition 6: Since $PC_\pi = \mathbf{31}$ and $\widehat{PC_\pi} \in \{\mathbf{13}, \mathbf{14}\}$ in C', the condition is satisfied.

Condition 20: Suppose `head`(`fragment`(TAIL)).PRED $\in$ {&INCS, &EXIT} in C. We have $\widehat{node_\pi}$.PRED = &CRASH, i.e., `head`(`fragment`($\widehat{node_\pi}$)).PRED = &CRASH. It follows that `fragment`($\widehat{node_\pi}$) $\neq$ `fragment`(TAIL) in C, which holds in C'. Therefore, the condition holds in C'.

30 (b). π executes Line **30** when $\widehat{PC_\pi} \notin \{\mathbf{13}, \mathbf{14}\}$.

In C, $PC_\pi = \mathbf{30}$ and $\widehat{PC_\pi} \notin \{\mathbf{13}, \mathbf{14}\}$. By Condition 6, $\widehat{PC_\pi} \in \{\mathbf{15}\} \cup \{\mathbf{25}, \mathbf{26}\}$. By Condition 10, $\widehat{node}_\pi$.Pred $\in \mathcal{N}'$. By Condition 2, $mypred_\pi = \widehat{node}_\pi$.Pred in C.

The **if** condition at Line **30** is met since $\widehat{node}_\pi$.Pred $\neq$ &Crash, therefore, π executes the Exit section of RLock. π then changes PC_π and $\widehat{PC_\pi}$ to **25**.

Condition 17: Applying Condition 17 to C, we have $\widehat{node}_\pi$.NonNil_Signal = `1` since $PC_\pi = \mathbf{30}$ in C. Therefore, the condition holds in C'.

31. π executes Line **31**.

In C $PC_\pi = \mathbf{31}$.

The step initializes $tail_\pi$ to Tail, the set V_π and E_π as empty sets, $tailpath_\pi$ to NIL, and $headpath_\pi$ to NIL. Since the invariant requires that i_π be between $[0, k]$ when $PC_\pi = \mathbf{32}$, we assume that the step implicitly initializes i_π to 0, although not noted in the code. Finally, the step sets PC_π to **32**.

Condition 21: Follows immediately from the description of the step above.

Condition 22: This condition follows immediately from Condition 16.

Condition 23: Since (V_π, E_π) are initialized to be empty sets, the condition follows.

Condition 24: Consider the fragments formed from the nodes pointed to by the cells in the Node array. If all the fragments have the Pred pointer of their head node to be in {NIL, &Crash}, then by definition of $\mathcal{Q}$ it is an empty set. Hence, Condition 24d holds. Otherwise, there is a fragment whose head node has its Pred pointer to be one of {&InCS, &Exit}. It follows that Condition 24c holds.

Condition 25: By Condition 12 it follows that $\forall \widehat{node}_{\pi'} \in \texttt{fragment}(\widehat{node}_\pi), \exists i \in [0, k-1], \text{Node}[i] = \widehat{node}_{\pi'}$. Therefore, the condition holds in C'.

Condition 26: Since (V_π, E_π) is an empty set in C', the condition holds vacuously.

Condition 27: All the conditions holds vacuously since the graph is empty and $i_\pi = 0$.

Condition 28: Suppose $\texttt{head}(\texttt{fragment}(tail_\pi))$.Pred $\in$ {&InCS, &Exit}. Since $\widehat{node}_\pi$.Pred = &Crash in C', it follows that the condition holds in C'.

Condition 29: Immediate from the description of the step above.

32 (a). π executes Line **32** when $i_\pi < k$.

In C, $PC_\pi = \mathbf{32}$ and $i_\pi < k$.

In this step the correctness condition of the **for** loop (i.e., $i_\pi \in [0, k-1]$) evaluates to *true* and PC_π is updated to **33**.

Since no shared variables are changed and no condition of the invariant is affected by the step, all the conditions continue to hold in C as they held in C'.

32 (b). π executes Line **32** when $i_\pi = k$.

In C, $PC_\pi =$ **32** and $i_\pi = k$.

In this step the correctness condition of the **for** loop (i.e., $i_\pi \in [0, k-1]$) evaluates to *false* and PC_π is updated to **39**.

Condition 28: From Condition 22 it follows that either $tail_\pi \in V_\pi \vee tail_\pi.\text{PRED} = \&\text{EXIT}$ in C'. In either case the condition holds in C'.

Condition 32: This follows immediately from Conditions 23, 25, 27 and the definition of fragment.

Condition 34: Follows immediately from Condition 24 and the fact that $i_\pi = k$.

Condition 35: Follows from Conditions 23, 24, 26, 27, the definition of fragment, and the fact that $\widehat{node_\pi}.\text{PRED} = \&\text{CRASH}$.

33. π executes Line **33**.

In C, $PC_\pi =$ **33**. By Condition 21, $i_\pi \in [0, k-1]$.

In this step, π sets cur_π to $\text{NODE}[i_\pi]$. It then updates PC_π to **35**.

Condition 30: $\text{NODE}[i_\pi]$ either has the value NIL or it does not. If it is the first case, we are done. In the second the condition follows from Condition 3.

34 (a). π executes Line **34** when $cur_\pi =$ NIL.

In C, $PC_\pi =$ **34**. and $cur_\pi =$ NIL.

Since the **if** is met, π is required to break the current iteration of the loop and start with its next iteration. Therefore, π increments i_π by 1 and changes PC_π to **32**.

Since no shared variables are changed and no condition of the invariant is affected by the step, all the conditions continue to hold in C as they held in C'.

34 (b). π executes Line **34** when $cur_\pi \neq$ NIL.

In C, $PC_\pi =$ **34** and $cur_\pi \neq$ NIL.

Since the **if** is not met, π changes PC_π to **35**.

Condition 30: The condition holds in C' as it held in C.

35. π executes Line **35**.

In C $PC_\pi =$ **35**.

The step executes cur_π.NonNil_Signal.wait() so that the procedure call returns when cur_π.NonNil_Signal = 1. The step also updates PC_π to **36** when it returns from the procedure call.

Condition 30: Since cur_π.NonNil_Signal = 1 as a result of the step, by Condition 5, cur_π.Pred $\in$ {&Crash, &InCS, &Exit} in C'. Therefore, the condition holds in C'.

36. π executes Line **36**.

In C PC_π = **36**.

The step sets $curpred_\pi$ to cur_π.Pred and updates PC_π to **37**.

Since no shared variables are changed and no condition of the invariant is affected by the step, all the conditions continue to hold in C as they held in C'.

37 (a). π executes Line **37** when $curpred_\pi \in$ {&Crash, &InCS, &Exit}.

In C, PC_π = **37** and $curpred_\pi \in$ {&Crash, &InCS, &Exit}.

The **if** condition at Line **37** is met, therefore, the step adds cur_π to the set V_π. It then increments i_π by 1 and updates PC_π to **32**.

Condition 23: Since the step adds only a vertex to the graph, the condition remains unaffected by the step.

Condition 24: If cur_π.Pred = &InCS, then Condition 24b is satisfied by the addition of cur_π to V_π. Otherwise, the condition holds as it held in C'.

Condition 26: If cur_π.Pred = &Crash, then the condition holds vacuously. Otherwise, cur_π.Pred $\in$ {&InCS, &Exit} and it follows that the condition holds in C'.

Condition 27: All sub-conditions are easy to argue, hence it follows that the condition holds in C'.

37 (b). π executes Line **37** when $curpred_\pi \notin$ {&Crash, &InCS, &Exit}.

In C, PC_π = **37** and $curpred_\pi \notin$ {&Crash, &InCS, &Exit}.

The **if** condition at Line **37** is not met, therefore, π updates PC_π to **38**.

Condition 28: Since $curpred_\pi \notin$ {&Crash, &InCS, &Exit}, $curpred_\pi$ was initialized from cur_π.Pred at Line **36**. It follows that $curpred_\pi \in \mathcal{N}'$. Therefore the condition holds in C'.

38. π executes Line **38**.

In C PC_π = **38**.

The step adds the elements cur_π and $curpred_\pi$ to the set V_π and the edge $(cur_\pi, curpred_\pi)$

to the set E_π. It then increments i_π by 1 and updates PC_π to **32**.

Condition 23: If $cur_\pi.\textsc{Pred} \notin V_\pi$ in C, then the condition holds in C'. Hence, assume $cur_\pi.\textsc{Pred} \in V_\pi$ in C (note, $cur_\pi.\textsc{Pred} = curpred_\pi$ in C). We have to argue that after the addition of the edge $(cur_\pi, curpred_\pi)$ in E_π, the graph (V_π, E_π) still remains directed and acyclic and the maximal paths in it remain disjoint in C'. During the configuration C, let σ_1 be the path in the graph (V_π, E_π) containing cur_π and σ_2 be the path containing $curpred_\pi$.

If $\sigma_1 \neq \sigma_2$, then the graph (V_π, E_π) continues to be directed and acyclic in C'. We argue that the maximal paths are disjoint as follows. Suppose for a contradiction that after adding the edge $(cur_\pi, curpred_\pi)$ there are two maximal paths σ and σ' that are not disjoint. It follows that this situation arises due to the addition of the edge $(cur_\pi, curpred_\pi)$, hence σ and σ' either share cur_π or $curpred_\pi$. Suppose they share cur_π as a common vertex. In C there is an edge $(cur_\pi, u) \in E_\pi$ (and therefore in the path σ_1) such that $u \neq curpred_\pi$. Applying Condition 27i to C, $cur_\pi.\textsc{Pred} = u$ or $cur_\pi.\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$ which is impossible since $cur_\pi.\textsc{Pred} = curpred_\pi$. Hence, assume that they share $curpred_\pi$ as a common vertex. It follows that there is an edge $(v, curpred_\pi) \in E_\pi$ appearing in the path σ_2 in the configuration C. By Condition 27c, $\exists i' \in [0, i_\pi - 1], v = \textsc{Node}[i']$ or $v.\textsc{Pred} = \&\textsc{Exit} \wedge \forall p' \in \mathcal{P}, \textsc{Node}[p'] \neq v$. If $\exists i' \in [0, i_\pi - 1], v = \textsc{Node}[i']$, then $\textsc{Node}[i'].\textsc{Pred} = \textsc{Node}[i_\pi].\textsc{Pred}$, which contradicts Condition 3. Otherwise, by Condition 27h, $\forall i' \in [0, k-1], \textsc{Node}[i'].\textsc{Pred} \neq curpred_\pi$, a contradiction (since $\textsc{Node}[i_\pi].\textsc{Pred} = curpred_\pi$ in C). Hence, it holds that if $\sigma_1 \neq \sigma_2$ in C, the maximal paths are disjoint in the graph in C'.

Otherwise, $\sigma_1 = \sigma_2$. It follows that $\texttt{start}(\sigma_1) = \texttt{start}(\sigma_2) = curpred_\pi$ and $\texttt{end}(\sigma_1) = \texttt{end}(\sigma_2) = cur_\pi$ (i.e., there is a path from $curpred_\pi$ to cur_π) in C. Applying Condition 27i inductively we see that $curpred_\pi.\textsc{Pred} \neq \&\textsc{Exit}$, otherwise it would imply $cur_\pi.\textsc{Pred} = \&\textsc{Exit}$. It follows by the contrapositive of Condition 27c that there is a distinct $i' \in [0, i_\pi - 1]$ for every vertex w in the path σ_1 such that $\textsc{Node}[i'] = w$ in C. That is, every vertex w in the path σ_1 is also a node $\widehat{node}_{\pi'}$ for some $\pi' \in \Pi$. However, since $cur_\pi.\textsc{Pred} \notin \{\text{NIL}, \&\textsc{Crash}, \&\textsc{InCS}, \&\textsc{Exit}\}$ (because there is a cycle with the presence of σ_1 and the pointer $cur_\pi.\textsc{Pred}$), we have a contradiction to Condition 4 since there is no $b \in \mathbb{N}$ for which the condition is satisfied. Therefore, $\sigma_1 \neq \sigma_2$ in C.

From this argument it follows that the condition holds in C'.

Condition 24: If Condition 24a holds in C, it continues to hold in C' and therefore the

condition is satisfied. Similarly for Condition 24b, because cur_π.Pred $\neq$ &Exit and if $cur_\pi = \texttt{end}(\sigma)$ for some maximal path which satisfied the condition, then it continues to satisfy the condition in C'. If $i_\pi < k-1$ and the condition held in C due to Condition 24c, then it continues to hold in C' for the new value of i_π. If $i_\pi = k-1$ and the condition held in C due to Condition 24c, it follows that Node$[k-1]$.Pred $\neq$ &Exit (by assumption above) and Node$[k-1]$.Pred.Pred $\in$ {&InCS, &Exit}. Therefore, by the step (Node$[k-1]$, Node$[k-1]$.Pred) is added as an edge in the graph and we have a path that satisfies Condition 24b in C'. If Condition 24d holds in C, then it holds in C' as well.

Condition 25: This condition holds by the definition of fragment and since the edge gets added to the graph.

Condition 26: It is easy to see that the second part of the condition holds because one of the nodes among v and v' was used up to enter the CS and hence even though the path runs through that node in the graph, the fragment is cut. Therefore, we argue the first part as follows. Suppose there is a $i < i_\pi$ such that Node$[i] = \widehat{node}$ for a $\widehat{node} \in \texttt{fragment}(v)$. In a previous iteration the node was added in the graph, and if $\widehat{node}$.Pred was an actual node, then it also got added to the graph along with an edge between them. Therefore, the condition holds in C'.

Condition 27: As argued above, cur_π.Pred $\in \mathcal{N}'$ (i.e., $curpred_\pi \in \mathcal{N}'$), it follows that $\forall v \in V_\pi, v \in \mathcal{N}'$ in C'. It is easy to see that the rest part of Condition 27a holds. Condition 27i holds because if cur_π.Pred $\in$ {&InCS, &Exit}, then the owner of $curpred_\pi$ already completed Line **28** to let the owner of cur_π into CS. Condition 27h holds from Condition 3. It is easy to see that the remaining sub-conditions hold in C'.

39. π executes Line **39**.

In C, $PC_\pi =$ **39**.

The step computes the maximal paths in the graph (V_π, E_π) and the set $Paths_\pi$ contains every such maximal path. The step then sets PC_π to **40**.

Condition 37: If there is a maximal path σ in (V_π, E_π) such that $\texttt{end}(\sigma)$.Pred $\in$ {&InCS, &Exit} and $\texttt{start}(\sigma)$.Pred $\neq$ &Exit, then the condition holds vacuously. Otherwise there is no maximal path σ for which $\texttt{end}(\sigma)$.Pred $\in$ {&InCS, &Exit} and $\texttt{start}(\sigma)$.Pred $\neq$ &Exit. By Condition 34, $\texttt{head}(\texttt{fragment}(tail_\pi))$.Pred $\in$ {&InCS, &Exit} $\vee$ $|\mathcal{Q}| = 0$ in C, which continues to hold in C'. Since $Paths_\pi$ is a set of all maximal paths in (V_π, E_π), there is no path $\sigma \in Paths_\pi$ for which $\texttt{end}(\sigma)$.Pred $\in$ {&InCS, &Exit} and $\texttt{start}(\sigma)$.Pred $\neq$ &Exit.

Therefore, the condition holds in C'.

40. π executes Line **40**.

In C, $PC_\pi = \mathbf{40}$. By Condition 27a, $\widehat{node}_\pi \in V_\pi$ and by Condition 2, $\widehat{node}_\pi = mynode_\pi$. By Condition 23b, all maximal paths in the graph are disjoint, therefore, every vertex in V_π appears in a unique path in $Paths_\pi$.

As argued above, $mynode_\pi \in V_\pi$ in C, therefore, there is a path σ in $Paths_\pi$ such that $mynode_\pi \in \sigma$. The step sets $mypath_\pi$ to be the unique path in $Paths_\pi$ in which $mynode_\pi$ appears. It then updates PC_π to **41**.

Condition 33: As argued above, in C' $mypath_\pi$ is the unique path in $Paths_\pi$ in which $mynode_\pi$ appears. Therefore, the condition holds in C'.

41. π executes Line **41**.

In C, $PC_\pi = \mathbf{41}$. By Condition 21, $tailpath_\pi = \text{NIL}$ in C. By Condition 23b, all maximal paths in the graph are disjoint, therefore, every vertex in V_π appears in a unique path in $Paths_\pi$.

In this step π checks if $tail_\pi \in V_\pi$. If so, it sets $tailpath_\pi$ to be the unique path in $Paths_\pi$ in which $tail_\pi$ appears. Otherwise, it just updates PC_π to **42**.

Condition 28: As argued above, the step sets $tailpath_\pi$ to be the unique path in $Paths_\pi$ in which $tail_\pi$ appears. Therefore, the condition holds in C'.

Condition 31: If $tailpath_\pi \neq \text{NIL}$ and $\texttt{end}(tailpath_\pi).\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\}$, it from Condition 27 that $\texttt{head}(\texttt{fragment}(tail_\pi)).\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\}$. Hence, the condition follows from Condition 29.

42 (a). π executes Line **42** when there is a path in $Paths_\pi$ not iterated on already.

In C, $PC_\pi = \mathbf{42}$ and there is a path in $Paths_\pi$ not iterated on already.

In this step π picks a path σ_π from $Paths_\pi$ that it didn't iterate on already in the loop on Lines **42-45**. It then sets PC_π to **43**.

Since no shared variables are changed and no condition of the invariant is affected by the step, all the conditions continue to hold in C as they held in C'.

42 (b). π executes Line **42** when there is no path in $Paths_\pi$ not iterated on already.

In C, $PC_\pi = \mathbf{42}$ and there is no path in $Paths_\pi$ not iterated on already.

In this step π finds that it has already iterated on all the paths from $Paths_\pi$ hence it just updates PC_π to **46**.

Since no shared variables are changed and no condition of the invariant is affected by the step, all the conditions continue to hold in C as they held in C'.

43. π executes Line **43**.

In C, $PC_\pi =$ **43**.

In this step, π checks if $\texttt{end}(\sigma_\pi).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$. If so, it updates PC_π to **44**; otherwise it updates PC_π to **42**.

Condition 36: If $PC_\pi =$ **44** in C', it is because of the **if** condition at Line **43** succeeded. From the description of the step given above, it follows that the condition holds in C'.

44. π executes Line **44**.

In C, $PC_\pi =$ **44**.

In this step, π checks if $\texttt{start}(\sigma_\pi).\textsc{Pred} \neq \&\textsc{Exit}$. If so, it updates PC_π to **45**; otherwise it updates PC_π to **42**.

Condition 36: If $PC_\pi =$ **45** in C', it is because of the **if** condition at Line **44** succeeded. Therefore, $\texttt{end}(\sigma_\pi).\textsc{Pred} \in \{\&\textsc{InCS}, \&\textsc{Exit}\}$ and $\texttt{start}(\sigma_\pi).\textsc{Pred} \neq \&\textsc{Exit}$. It follows that the length of the path is more than 1 and $\texttt{start}(\sigma_\pi) = \texttt{tail}(\texttt{fragment}(\texttt{start}(\sigma_\pi)))$. Therefore, from the description of the step given above, it follows that the condition holds in C'.

45. π executes Line **45**.

In C, $PC_\pi =$ **45**.

The step sets $headpath_\pi = \sigma_\pi$ and updates PC_π to **46**.

Condition 38: This condition follows as a result of Conditions 1, 34, 35 and since the step sets $headpath_\pi = \sigma_\pi$.

46 (a). π executes Line **46** when $tailpath_\pi \neq \text{NIL} \wedge \texttt{end}(tailpath_\pi).\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\}$.

In C, $PC_\pi =$ **46** and $tailpath_\pi \neq \text{NIL} \wedge \texttt{end}(tailpath_\pi).\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\}$.

In this step π checks for the **if** condition at Line **46** to be met. Since $tailpath_\pi \neq \text{NIL} \wedge \texttt{end}(tailpath_\pi).\textsc{Pred} \notin \{\&\textsc{InCS}, \&\textsc{Exit}\}$, the **if** condition is not met, hence, π updates PC_π to **48**.

Since no shared variables are changed and no condition of the invariant is affected by the

step, all the conditions continue to hold in C as they held in C'.

46 (b). π executes Line **46** when $tailpath_\pi = \text{NIL} \vee \texttt{end}(tailpath_\pi).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}$.

In C, $PC_\pi =$ **46** and $tailpath_\pi = \text{NIL} \vee \texttt{end}(tailpath_\pi).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}$.

In this step π checks for the **if** condition at Line **46** to be met. Since $tailpath_\pi = \text{NIL} \vee \texttt{end}(tailpath_\pi).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}$, the **if** condition is met, hence, π updates PC_π to **47**.

Since no shared variables are changed and no condition of the invariant is affected by the step, all the conditions continue to hold in C as they held in C'.

47. π executes Line **47**.

In C, $PC_\pi =$ **47**.

In this step π performs a FAS on TAIL with the node $\texttt{start}(mypath_\pi)$ and stores the returned value of the FAS into $mypred_\pi$. It then updates $\widehat{PC_\pi}$ to **14** and PC_π to **49**.

Condition 39: By Condition 28, $\texttt{fragment}(\widehat{node}_\pi) \neq \texttt{fragment}(\text{TAIL})$ in C. Applying Condition 16 to C and TAIL we see that the condition holds in C'.

48. π executes Line **48**.

In C, $PC_\pi =$ **48**.

If $headpath_\pi \neq \text{NIL}$ then the step sets $mypred_\pi = \texttt{start}(headpath_\pi)$; otherwise it sets $mypred_\pi = \&\text{SPECIALNODE}$. The step then updates $\widehat{PC_\pi} =$ **14** and $PC_\pi =$ **49**.

Condition 39: For any value that $mypred_\pi$ takes in the step, we note that $\texttt{head}(\texttt{fragment}(mypred_\pi)).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}$. If $headpath_\pi \neq \text{NIL}$, then all the parts of the condition are satisfied in C' which can be verified from Condition 38 holding in C. Also, $\texttt{fragment}(\widehat{node}_\pi) \neq \texttt{fragment}(\text{first}(headpath_\pi))$ in C', since $\texttt{head}(\texttt{fragment}(\text{first}(headpath_\pi))).\text{PRED} \in \{\&\text{INCS}, \&\text{EXIT}\}$.

If $headpath_\pi = \text{NIL}$ then $mypred_\pi = \&\text{SPECIALNODE}$ in C' and it is easy to verify again that the condition holds in C'. Therefore, the condition holds in C'.

49. π executes Line **49**.

In C $PC_\pi =$ **49**.

As a result of the step, $mynode_\pi.\text{PRED}$ to $mypred_\pi$ and updates $\widehat{PC_\pi}$ to **25** in C'. π also executes the Exit section of RLOCK, hence $PC_\pi =$ **25** in C'.

The argument for correctness for this step is similar to that of the argument given for execution of Line **14**. Therefore, we refer the reader to those arguments above.

Crash. π executes a crash step.

This step changes PC_π to **10** and sets the rest of the local variables to arbitrary values. The values of the shared variables remain the same as before the crash.

The step does not affect any condition, so the invariant continues to hold in C'.

Thus, by induction it follows that the invariant holds in every configuration of every run of the algorithm. □

Part III

Lock-Free Data Structures

Chapter 7

Concurrent Union Find

7.1 Introduction

As data sets get bigger and bigger, it becomes more and more important to harness the potential of parallelism to solve computational problems - even linear time is too slow. In the late twentieth century, many beautiful and efficient algorithms were developed in the PRAM (parallel random access machine) model, which assumes a memory shared among many synchronized processors. In practice, however, synchronization is expensive or may not be possible. A weaker model that has attracted much attention in the distributed systems community is the APRAM (asynchronous parallel random access machine) model, in which a common memory is shared among many unsynchronized processors. In the most general version of this model, any processor can be arbitrarily slow compared to any other.

Obtaining efficiency bounds in the APRAM model is extremely challenging: the use of locks, for example, seems to make it impossible to guarantee efficiency, since one process could set a lock and then go to sleep indefinitely, blocking progress by any other process that needs access to the same resource. To overcome this problem, systems researchers have invented synchronization primitives that do not use locks, notably CAS (compare and swap) [89], transactional memory [90], and others. These primitives allow at least the possibility of obtaining good efficiency bounds for asynchronous concurrent algorithms. Yet, except for "embarrassingly parallel" computations, this possibility is almost unrealized. Indeed, we know of only one example of a concurrent data structure (other than our work, to be described) for which a work bound without a term at least linear in the number of processes has been obtained. This is an implementation by Ellen and Woefel [57] of a fetch-and-increment object.

An important problem in data structures that could benefit from an efficient concurrent algorithm is *disjoint set union*, also known as the *union-find* problem. The simplest version of this problem requires maintaining a collection of disjoint sets, each containing a unique element called its *leader*, under two operations:

find(x): return the leader of the set containing element x.

unite(x, y): if elements x and y are in different sets, unite these sets into a single set and designate some element in the new set to be its leader; otherwise, do nothing.

Each initial set is a singleton, whose leader is its only element. Note that the implementation is free to choose the leader of each new set produced by a *unite*. This freedom simplifies concurrent implementation, as we discuss in Section 7.4. Other versions of the problem add operations for initializing singleton sets and for maintaining and retrieving information about the sets such as names or sizes. We study the simplest version but comment on extensions in Section 7.10.

Applications of sequential disjoint set union include storage allocation in compilers [136], finding minimum spanning trees using Kruskal's algorithm [134], maintaining the connected components of an undirected graph under edge additions [198, 51, 95], testing percolation [181], finding loops and dominators in flow graphs [194, 193, 63], and finding strong components in directed graphs. Some of these applications, notably finding connected components [192, 184, 121, 86, 173, 144] and finding strong components, are on immense graphs and could potentially benefit from the use of concurrency to speed up the computation. For example, model checking requires finding strong components in huge, implicitly defined directed graphs [209, 22, 24]. There are sequential linear-time strong components algorithms [192, 183], but these may not be fast enough for this application. The sequential algorithms use depth-first search [192], which apparently cannot be efficiently parallelized [175]. If one had an efficient concurrent disjoint set union algorithm one could use it in combination with breadth-first search to potentially speed up model checking. This application, described to the second author by Leslie Lamport, was the original motivation for our work.

The classical sequential solution to the disjoint set union problem is the *compressed tree* data structure [67, 62, 97, 198, 196]. With appropriate tree linking and path compaction rules, m operations on sets containing a total of n elements take $O(\alpha(n, m/n))$ time [198, 196, 75], where α is a functional inverse of Ackermann's function, defined in Section 7.3. Three linking rules that suffice are linking by size [198], linking by rank [196], and linking by random index [75]; three compaction rules that suffice are compression [198, 196, 75], splitting [196, 75], and halving [196, 75].

Perhaps surprisingly, there has been almost no previous research on wait-free concurrent disjoint set union. We have found only one such effort, that of Anderson and Woll [12]. Their work contains a number of significant ideas that are the genesis of our results, but it has many flaws that reveal the subtlety of the problem. We use their concurrency model. In one of our linking algorithms we use DCAS (double compare and swap), as a synchronization primitive, whereas they used only the weaker CAS (compare and swap) primitive.

Anderson and Woll considered an alternative formulation of the problem in which sets do not have leaders and the two operations are *same-set*(x, y), which returns true if x and y are in the same set and false otherwise, and *unite*(x, y), which combines the sets containing x and y into a single set if these sets are different. (We discuss *same-set* further in Section 7.4.) They attempted to develop an efficient concurrent solution that combines linking by rank with a concurrent version of path halving. They claimed a bound of $O(m \cdot (p + \alpha(m, 1))$ on the total work, where p is the number of processors. (They did not treat n as a separate parameter.). Their linking method can produce paths of $\Omega(p)$ nodes of equal rank. The $O(mp)$ term in their work bound accounts for such paths. Their proof of their upper bound is not correct, because they did not consider interference among different processes doing halving on intersecting paths. Whether or not their bound is correct, it is easy to show that their algorithm can take $\Omega(np)$ work to do $n - 1$ unite operations, compared to the $O(n\alpha(n, 1))$ time required by one process. Thus in the worst case their work bound gives essentially no speedup.

Anderson and Woll also claimed a work bound of $O(m \cdot (\alpha(m, 1) + \log^* p))$ for a synchronous PRAM algorithm that uses deterministic coin tossing [36] to break up long paths of equal-rank nodes. They provided no details of this algorithm and no proof of the work bound. We think that their bound is incorrect and that the work bound of their algorithm is $\Omega(n \log p)$, since it is easy to construct sets of operations that do linking by rank exactly but such that concurrent finds with halving take $\Omega(\log p)$ steps per find, even on a PRAM. See Section 7.9. Deterministic coin tossing does seem to be a good idea, however: we conjecture that it can give an efficient (but complicated) deterministic set union algorithm in the APRAM model using only CAS for synchronization, at the cost of a multiplicative $\log^* p$ factor in the work bound.

In this chapter, we apply the ideas of Anderson and Woll and some additional ones to develop several efficient concurrent algorithms for disjoint set union. We give three concurrent implementations of *unite*, one deterministic and the other two randomized. The deterministic method uses DCAS to do linking by rank. The randomized methods use only CAS: one does linking by random

index, the other does randomized linking by rank. We also give two concurrent implementations of path splitting, *one-try* and *two-try splitting*. The former is simpler, but we are able to prove slightly better bounds for the latter, bounds that we think are tight for the problem.

We prove that any of our linking methods in combination with one-try splitting does set union in $O\left(m \cdot \left(\log\left(\frac{np^2}{m}+1\right)+\alpha\left(n,\frac{m}{np^2}\right)\right)\right)$ work, and in combination with two-try splitting in $O\left(m \cdot \left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n,\frac{m}{np}\right)\right)\right)$ work. Each set operation takes $O(\log n)$ steps. These bounds are worst-case for deterministic linking and high-probability for randomized linking. The $O(\log n)$ step bound per operation holds even without path splitting; without splitting, the work bound is $O(m \log n)$. The work and step bounds for randomized linking by rank hold even for an adversarial scheduler, provided that scheduling is based only on information sent to the scheduler, or we allow a form of CAS that writes a random bit. The work and step bounds for linking by random index hold provided that the randomization is independent of the order in which the unite operations are executed, or, more precisely, independent of the "linearization order" of the unite operations. (We define linearization order in Section 7.2.) We also show that $\Omega\left(m \cdot \left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n,\frac{m}{np}\right)\right)\right)$ work is needed in the worst case for any algorithm satisfying a symmetry assumption, which implies that our work bound for two-try splitting is best possible for such algorithms.

Our work is theoretical, but others [3, 51, 95] have implemented some of our algorithms on CPUs and GPUs and experimented with them. On many realistic data sets, our algorithms run as fast or faster than all others.

The remainder of our chapter contains 8 sections. Section 7.2 describes our concurrency model. Section 7.3 describes the compressed tree data structure and sequential algorithms for disjoint set union. Section 7.4 presents concurrent linking by index, a special case of which is concurrent linking by random index, and one-try and two-try splitting. Section 7.5 presents preliminary versions of deterministic and randomized linking by rank. These versions rely on some simplifying assumptions that we eliminate in Section 7.6. Section 7.8 gives upper bounds on the total work of our algorithms. Section 7.9 presents lower bounds. Section 7.10 contains some final remarks and open problems.

7.2 Concurrency Model

Our concurrency model is the same as that of Anderson and Woll: a shared memory multiprocessor, otherwise known as an asynchronous random-access machine (APRAM). We assume that p processes run concurrently but asynchronously, each doing a different set operation. Each process

has a private memory. In addition, all processes have access to a shared memory that supports concurrent reads but not concurrent writes.

To provide synchronization of writes to shared memory, we use the *compare and swap* primitive CAS(x, y, z). Given the address x of a block of shared memory and two values y and z, this operation tests whether block x holds value y; if so, it stores value z in block x (overwriting y) and returns true; if not, it returns false. We also consider the two-block extension DCAS(u, v, w, x, y, z). Given the addresses u and x of two blocks of shared memory and four values v, w, y, and z, this operation tests whether block u holds value v and block x holds value y; if both are true, it stores value w in block u and value z in block x and returns true; if not, it returns false. These operations are atomic: once one starts, it completes before any other operation can read, write, CAS, or DCAS the affected block or blocks. Although both CAS and DCAS return a value indicating success or failure, many of our algorithms do not actually use these values.

In one version of our randomized linking algorithm we use the following randomized version of CAS: atomic operation CAS(x, y, \$) tests whether the value of x is y and, if so, sets the value of x equal to true or false, each with probability $1/2$. Such a randomized atomic write operation has been used in algorithms for achieving consensus [34].

Many current hardware designs include CAS as an instruction; DCAS was supported on the Motorola 68030 [162] but not on any current hardware, as far as we know. As we demonstrate in Section 7.5, it is straightforward to implement linking by rank using DCAS, but much harder using only CAS.

We study concurrent algorithms for disjoint set union that are *linearizable* [93] and *bounded wait-free* [89]. To be linearizable means that (i) the outcome of a concurrent execution is the same as if each set operation were executed instantaneously at some distinct time (its linearization time) during its actual execution and (ii) the sequential execution sequence given by the linearization times is correct; that is, all find operations produce answers that are correct at their linearization times. The linearization times define a total order of the operations, called the *linearization order*. Although we focus on linearizable algorithms, some applications of disjoint set union may not require linearizability for correctness. We briefly discuss this issue in Section 7.10, and leave further investigation as an open problem.

To be bounded wait-free means that every operation finishes in a bounded number of its own steps. The *total work* done by a concurrent solution is the total number of steps done by all processes to complete all operations.

Two weaker progress properties than bounded wait-freedom are *wait-freedom* and *lock-freedom* [91]. A concurrent solution is wait-free if every process is guaranteed to finish. It is lock-free if every operation can execute its next step when it chooses to do so, and at least one process is guaranteed to finish its operation. In general a lock-free solution need not be wait-free, and a wait-free solution need not be bounded wait-free. In our version of disjoint set union, the number of elements is fixed, which makes it easy to guarantee bounded wait-freedom. This remains true if we add an operation that allows the creation of singleton sets containing new elements, as long as the total number of set operations is bounded. If we allow an unbounded number of singleton sets to be created, then our solutions are no longer wait-free, but they remain lock-free. In this case there are no meaningful work bounds.

7.3 Data Structure and Sequential Algorithms

Our concurrent disjoint set union algorithms use the same data structure as the best sequential algorithms: a *compressed forest*. This forest contains one rooted tree per set, whose nodes are the elements of the set and whose root is the set leader. Each node x has a pointer $x.p$, to its parent if it has a parent or to itself if it is a root. The root of the tree is the leader of the set.

In this section we explain the sequential set union algorithms. We present pseudo-code for the *same-set* and *find* procedures as Algorithm 9, and procedures for *unite* and its helper-method *link* as Algorithm 10. The sequential algorithm pseudo-code we present is not optimized for brevity. Rather, we take care to present pseudo-code that is as similar to the forthcoming concurrent algorithms as possible, thereby highlighting the key observations and distinctions that arise in the concurrent code.

The sequential algorithm for *find*(x) follows parent pointers from x until reaching a node u that points to itself, optionally *compacts* the find path (the path of ancestors from x to u) by replacing the parent of one or more nodes on the find path by a proper ancestor of its parent, and returning u. *Naïve find* does no compaction. Three good compaction rules are *compression*, *splitting*, and *halving*. Compression replaces the parent of every node on the find path by the root u. Splitting replaces the parent of every node on the find path by its grandparent. Halving replaces the parent of every other node on the find path by its grandparent, starting with x. Figure 7.3.1 illustrates how these algorithms restructure a path of nodes when *find* is called on node 8, the bottom-most node.

The sequential implementation of *same-set*(x, y) does *find*(x) and *find*(y), returning the roots u and v of the trees containing x and y, respectively, and returns true if $u = v$, false otherwise. Algorithm 9 is the pseudo-code for *same-set* and the variations of *find*.

Algorithm 9 Sequential same-set algorithm with alternative implementations of find. The pseudo-code is written to match the forthcoming concurrent version as closely as possible, so that the key differences are more clear.

```
1:  procedure same-set(x, y)
2:      u ← find(x)
3:      v ← find(y)
4:      return u = v

5:  procedure findNaïve(x)
6:      u ← x; v ← u.p
7:      while v ≠ u do
8:          u ← v; v ← u.p
9:      return v

10: procedure findCompress(x)
11:     root ← findNaïve(x)
12:     u ← x
13:     while u ≠ root do
14:         v ← u.p; u.p ← root; u ← v
15:     return root

16: procedure findSplit(x)
17:     u ← x; v ← u.p; w ← v.p
18:     while v ≠ w do
19:         u.p ← w; u ← v; v ← u.p; w ← v.p
20:     return v

21: procedure findHalve(x)
22:     u ← x; v ← u.p; w ← v.p
23:     while v ≠ w do
24:         u.p ← w; u ← w; v ← u.p; w ← v.p
25:     return v
```

The sequential implementation of *unite*(x, y) does *find*(x) and *find*(y), returning the roots u and v of the trees containing x and y, respectively, and tests whether $u = v$. If $u \neq v$, it *links* u and v by making one the parent of the other. Three good linking rules are *linking by size*, *linking by rank*, and *linking by random index*. *Linking by size* maintains the *size* (number of nodes) of each tree in its root, and makes the root of the tree of larger size the parent of the other, breaking a tie arbitrarily. *Linking by rank* maintains a non-negative integer *rank* for each root, initially zero, and makes the root of larger rank the parent of the other, breaking a tie by adding one to the rank of one of the roots. In the pseudo-code, we use $u.r$ to represent node u's rank. *Linking by index* chooses a fixed total order of the nodes and makes the root of larger index the parent of the other. *Linking by random index* is the special case of linking by index that chooses the total order of nodes uniformly at random. Algorithm 10 is the pseudo-code for *unite* and the variations of *link*.

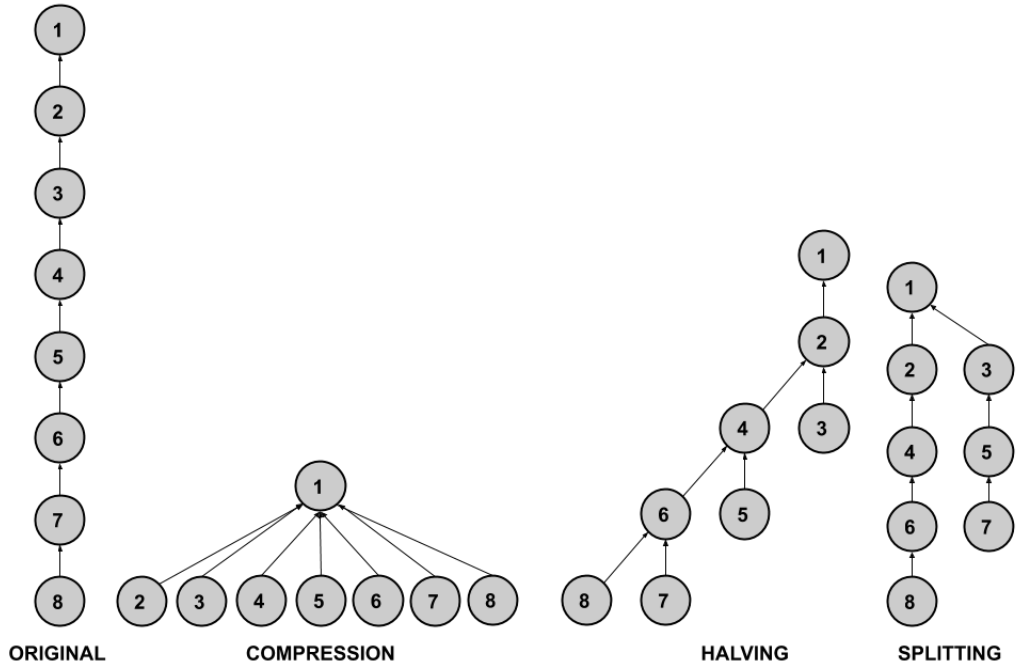


Figure 7.3.1: The results of running *find*(8) on the original path with the three different types of compaction: compression links all the nodes on the find path directly to the root, thereby "compressing" the path; halving links alternating nodes on the find path to their grandparents, thereby creating a path of "half" the length with nodes hanging off; splitting links every node on the find path to its grandparent, thereby "splitting" one path in two.

Algorithm 10 Sequential unite algorithm, with multiple implementations of link. The pseudocode is written to match the forthcoming concurrent version as closely as possible so that the key differences are more clear.

```
 1: procedure unite(x, y)
 2:     u ← find(x)
 3:     v ← find(y)
 4:     if u ≠ v then link(u, v)

 5: procedure linkByRank(u, v)
 6:     r ← u.r; s ← v.r
 7:     if r < s then u.p ← v
 8:     else if r > s then v.p ← u
 9:     else
10:         v.r ← v.r + 1
11:         u.p ← v

12: procedure linkByIndex(u, v)
13:     if u < v then u.p ← v
14:     else v.p ← u

15: procedure linkBySize(u, v)
16:     if u.size ≤ v.size then
17:         u.p ← v
18:         v.size ← v.size + u.size
19:     else
20:         v.p ← u
21:         u.size ← u.size + v.size
```

Linking by size, rank, or random index combined with naïve find, compression, splitting or halving gives an algorithm that takes $O(\log n)$ time for an operation on a set or sets containing n elements, worst-case for deterministic linking, high-probability for linking by random index. Use of compaction improves the amortized time per operation: any combination of compression, splitting, or halving with linking by size, rank, or random index gives an algorithm that takes $O(m \cdot \alpha(n, m/n))$ time to do m operations on sets containing a total of n elements. The bound is worst-case for linking by size or rank, average-case for linking by randomized index. Here is a functional inverse of Ackermann's function defined as follows. We recursively define $A_k(n)$ for non-negative integers k and n as follows:

$A_0(n) = n + 1$; $A_k(0) = \mathrm{A}_{k-1}(1)$ if $k > 0$; $A_k(n) = A_{k-1}(A_k(n-1))$ if $k > 0$ and $n > 0$.

For a non-negative integer n and non-negative real-valued d,

$$\alpha(n, d) = \min\{k > 0 \mid A_k(\lfloor d \rfloor) > n\}$$

Lemma 7.3.1. *$A_k(n) < \min\{A_{k+1}(n), A_k(n+1)\}$, i.e., $A_k(n)$ is strictly increasing in k and n.*

Proof. The proof is by double induction on k and n. $A_0(n) = n + 1 < n + 2 = A_0(n + 1)$, and $A_0(0) = 1 < 2 = A_1(0)$. Let $k > 0$. Suppose the lemma holds for $k' < k$ and all n. Then $A_k(0) < A_k(0) + 1 = A_0(A_k(0))\ A_{k-1}(A_k(0)) = A_k(1) = A_{k+1}(0)$. Thus the lemma also holds for k and $n = 0$. Let $k > 0$ and $n > 0$. Suppose the lemma holds for $k' < k$ and all n, and for k and $n - 1$. Then $A_k(n) < A_k(n) + 1 = A_0(A_k(n))\ A_{k-1}(A_k(n)) = A_k(n + 1)$, and $A_k(n) = A_k(A_0(n - 1)) < A_k(A_{k+1}(n - 1)) = A_{k+1}(n)$. □

Corollary 7.3.2. *$\alpha(n, d)$ is non-decreasing in n and non-increasing in d.*

Our goal is to extend at least one sequential set union algorithm to the concurrent model of Section 7.2 and to obtain an almost-linear work bound that grows sublinearly with p, the number of processes. For convenience in stating bounds, we assume that $2 \le p \le n \le m$, and that there is at least one unite of different elements. We denote the base-two logarithm by lg.

7.4 Concurrent Linking and Splitting

Concurrency significantly complicates the implementation of the set operations. One complication is that processes can interfere with each other by trying to update the same field at the same time, requiring our algorithms to be robust to such interference. Consider doing unites concurrently.

To do *unite*(x,y), we can start as in the sequential case by finding the roots u and v of the trees containing x and y, respectively. Then we can try to link u and v by doing a CAS to make v the parent of u or vice-versa. But we must allow for the possibility that the CAS can fail, for example if it tries to make v the parent of u but in the meantime some other process makes another node the parent of u. If this happens we must retry the unite. When retrying, we start the new finds at u and v rather than at x and y, to avoid revisiting nodes. Anderson and Woll [12] proposed this method; the following pseudocode implements it. Method *link*(u, v), to be defined, tries to make one of two roots u and v the parent of the other.

Algorithm 11 : Concurrent unite algorithm.

```
1: procedure unite(x, y)
2:     u ← find(x); v ← find(y)*
3:     while u ≠ v do
4:         link(u, v)*
5:         u ← find(u); v ← find(v)*
```

In this and subsequent implementations, asterisks denote linearization points. The linearization point of a unite is the linearization point of the successful link if there is one, or the linearization point of the last find if no link is successful.

Concurrency also imposes constraints on the linking rule. We need to prevent concurrent links from creating a cycle of parent pointers other than a loop at a root. For example, three concurrent links might make v the parent of u, w the parent of v, and u the parent of w. The simplest way to prevent such cycles is to do linking by index, which we can implement using CAS. We denote the total order of nodes by "$<$" . The following pseudocode implements linking by index:

Algorithm 12 : Concurrent linking by index algorithm.

```
1: procedure link(u, v)
2:     if u < v then CAS(u.p, u, v)*
3:     else CAS(v.p, v, u)*
```

The linearization point of the link is its CAS. A link is *successful* if its CAS returns true. For any total order, linking by index guarantees acyclicity. *Linking by random index* is the special case of linking by index that chooses the total order uniformly at random.

With this implementation of link, a link can succeed even though the new parent itself becomes a child of another node at the same time. Fortunately this affects neither correctness nor efficiency.

We could prevent this anomaly by using DCAS to do links, which allows us to guarantee that the new parent remains a root. But this has two drawbacks. First, it uses DCAS, whereas our goal is to use only CAS if possible. Second, if all links are done using DCAS, the total work can be linear in p, as we discuss in Section 7.5.1.

Next we consider finds. Concurrent naïve finds do not interfere with each other, since such finds do not change the data structure. Thus we can do such finds exactly as in the sequential case. The following pseudocode implements concurrent naïve find:

Algorithm 13 : Concurrent Naïve find algorithm.

```
1: procedure find(x)
2:     u ← x; v ← u.p*
3:     while v ≠ u do
4:         u ← v; v ← u.p*
5:     return u
```

The linearization point of a find is the last update of v.

Concurrent finds with compaction *can* interfere with each other. Consider a sequential find with splitting. Let u be the current node visited by the find. One step of the find consists of setting $v = u.p$; setting $w = v.p$; and, if $v \neq w$, replacing $u.p$ by w and then setting $u = v$. Steps continue until $v = w$, when the find finishes by returning v. The only update to the data structure in a step is the replacement of $u.p$ by w. We obtain a concurrent version of splitting by using CAS($u.p$, v, w) to do the update. The following pseudocode implements this method, which we originally presented in [118] and which is based on Anderson and Woll's version of find with halving:

Algorithm 14 : Concurrent Find with One-Try Splitting algorithm.

```
1: procedure find(x)
2:     u ← x; v ← u.p; w ← v.p*
3:     while v ≠ w do
4:         CAS(u.p, v, w); u ← v; v ← u.p; w ← v.p*
5:     return v
```

The linearization point of a find is the last update of w. We call this method *one-try splitting* because it tries once to update $u.p$ and then changes the current node from u to v, whether or not the update of $u.p$ has succeeded.

Concurrent splits can produce anomalies that are not possible if splits are sequential, as a simple

example shows. (See Figure 2.) Suppose a, b, c, d, e is a path in a tree built by linking by index, and that four processes, 1, 2, 3, and 4 begin concurrent finds with one-try splitting starting at a, a, b, and b, respectively. We denote the local variables of process i by u_i, v_i, w_i. First, process 1 sets $u_1 = a$, $v_1 = a.p = b$, and $w_1 = b.p = c$. Second, process 3 sets $u_3 = b$, $v_3 = c$, $w_3 = d$, and replaces $b.p$ by d. Third, process 4 sets $u_4 = b$, $v_4 = d$, $w_4 = e$, and replaces $b.p$ by e. Fourth, process 2 sets $u_2 = a$, $v_2 = b$, and $w_2 = d$. Fifth, process 1 replaces $a.p$ by c. Sixth, process 2 attempts to replace $a.p$ by d but fails, because process 1 changed $a.p$ after process 2 read it. Observe that just before process 1 replaces $a.p$ by c, c is not an ancestor of a, even though it was when process 1 read it. This threatens correctness. Furthermore, even though the failure of process 2 to update $a.p$ guarantees that $a.p$ has changed since process 2 read it, the new value of $a.p$, namely c, is not an ancestor of the current grandparent of a, namely e, violating a property used in the analysis of sequential splitting. Finally, even though the new parent c of a is higher in index than the old parent b of a (as we prove in Theorem 7.4.1), the new grandparent d of a is *lower* than the old grandparent e of a.

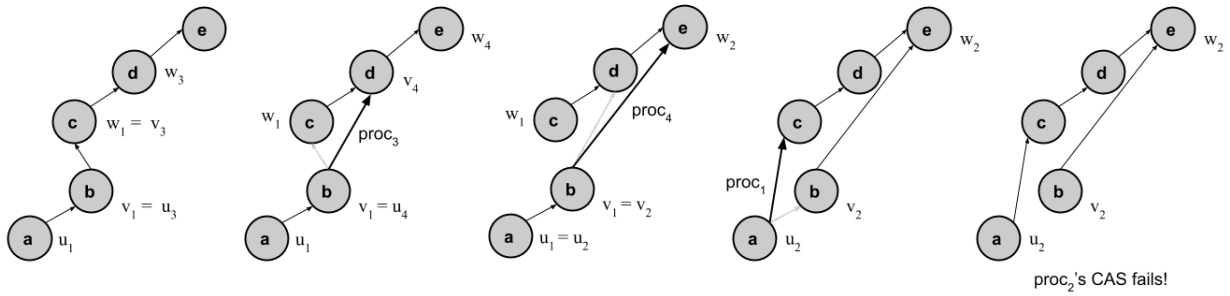


Figure 7.4.1: Interference in concurrent splitting: process 1 updates a's parent to a node that is not its ancestor; process 2's CAS fails. These difficulties do not occur in the sequential setting.

Fortunately, correctness requires only a weak property of compaction, one that holds for one-try splitting and many other methods. We introduce an analytic tool called the *union forest* in order to explain the property. We assume that if a compaction changes the parent $w.p$ of a node w by a CAS, $w.p \neq w$ just before the change; that is, w is not a root. Equivalently, only a link can change the parent of a root. Suppose we do linking by index. Consider a fixed history, i.e. a concurrent execution of several *unite*, *same-set*, and *find* operations by different processes up to some time t. For this fixed history, the *union forest* is the set of trees such that the parent of a node w is the *first* value, other than w, that $w.p$ takes on during the history; if $w.p = w$ throughout the history, then w is a root in the union forest.

Claim 1. The union forest is a forest.

Proof. Since linking is by index, when a link changes the parent of a root w from w to z, $z > w$. Hence the union forest contains no cycles of parent pointers other than loops. Thus the union forest is indeed a forest. □

We call a compaction method *valid* if it visits nodes on a single path in the union forest, each vertex visit takes $O(1)$ steps, each replacement of a parent w by another node z (of which there may be none) is such that z is a proper ancestor of w in the union forest, and the linearization point of the find doing the compaction is the last read of a parent that returns the node itself. The parent update requirements are only with respect to the fixed union forest, *not* with respect to the dynamically changing actual forest maintained by the data structure. In particular, although find with splitting can change the parent $w.p$ of a node w to a non-ancestor of w in the actual forest (see Figure 7.4.1), it cannot do so in the union forest. Indeed, splitting is valid.

The following theorem states the correctness of linking by index with finds that do valid compactions.

Theorem 7.4.1. *Any disjoint set union algorithm that does linking by index in combination with finds that do valid compaction is linearizable. The parent of any non-root node has higher index than the node, and the parents define a set of trees that partition the nodes into the correct disjoint sets. Furthermore each set operation stops in $O(h)$ steps, where h is the height (maximum number of edges on a path) of the union tree, so the algorithm is bounded wait-free.*

Proof. An induction on the number of parent changes using the transitivity of "$<$" shows that the parent of any node never has smaller index than the node. This implies that the only cycles are loops at roots. Parent changes done during compactions do not change the node partition defined by the trees. A link that makes v the parent of u must be such that u is a root before the link, $u < v$, and u and v are in the trees containing the two nodes x and y that are the inputs to the unite that does the link. It follows by induction on the number of parent changes that at all times the trees correctly partition the nodes: a find cannot change this partition, and a link unites the trees containing the nodes that are the inputs to the corresponding unite. Correctness of the linearization points follows in a straightforward way by induction on the number of parent changes: When a find reads the parent of a root, that root at that moment is the leader of the set containing the input to the find; when a unite does a link, the partition remains correct; when a test "$u \neq v$" in unite returns false, the inputs to the unite are in the same set.

Since the nodes visited during a find are on a single path in the union forest, and each node visit takes $O(1)$ steps, each find stops in $O(h)$ steps. (Our assumption that there is at least one unite of different elements implies $h > 0$.) The nodes visited during a unite are on two paths in the union forest. Consider the node visits in the order they occur. Each node visit takes $O(1)$ steps, but a node can be visited many times. This can only happen while it is a root; once it becomes a child, it can only be visited once more (as the input to a find). Consider the nodes u and v just before an execution of the test "$u \neq v$" in unite. Each of u and v was a root at some time during the find that computed it. If the test "$u \neq v$" succeeds, whichever of u and v is smaller in the total order will be a child after the next link (whether or not the link succeeds). Suppose without loss of generality it is u. We charge the next visits to u and v to u becoming a child. There are at most $2h$ such events. It follows that the total number of node visits during the unite, and hence the total number of steps, is $O(h)$. □

Having dealt with correctness, we discuss concurrent compaction in more detail. The monotonicity of parents (each new parent is higher in index than the old one) allows us to extend the analysis of sequential splitting to one-try splitting, although the extension is not straightforward. On the other hand, the analysis of sequential halving relies on monotonicity of grandparents, which fails in the concurrent setting, as our example above shows. Anderson and Woll [12] claimed a good work bound for their concurrent version of halving, but they overlooked the problem of non-monotonicity. We see no way to get a good work bound for their method.

Even though we can prove good efficiency bounds for one-try splitting, we can prove slightly better bounds for a related compaction method that tries to change each parent pointer twice instead of once. We call this method *two-try splitting*. The following pseudocode implements find with two-try splitting:

Algorithm 15 : Concurrent Find with Two-Try Splitting algorithm.

```
1: procedure find(x)
2:     u ← x; v ← u.p; w ← v.p*
3:     while v ≠ w do
4:         CAS(u.p, v, w); v ← u.p; w ← v.p
5:         CAS(u.p, v, w); u ← v; v ← u.p; w ← v.p*
6:     return v
```

The linearization point of a find is the last assignment to w. If every attempted parent change succeeds, the effect of a single two-try split is to replace the parent of every other node on the find

path by its great-grandparent. This splits the original path into two paths, each containing half the nodes on the original path, but the split is different from that produced by one-try splitting: if the nodes on the original path are numbered consecutively from 1, the latter produces a path of nodes 1, 3, 5, 7... and another path of nodes 2, 4, 6, 8... ; the former produces a path of nodes 1, 4, 5, 8, 9... and another path of nodes 2, 3, 6, 7, 10, 11...

A variant that has the same work bounds as two-try splitting is *conditional two-try splitting*, in which the second try occurs only if the first one fails. We omit a detailed discussion of this variant, since its pseudocode is a bit longer and it is unclear whether avoiding extra parent changes improves efficiency.

Both one-try and two-try splitting are valid compaction methods, so Theorem 7.4.1 holds for both of them.

We conclude this section by presenting Anderson and Woll's concurrent implementation of *same-set*, which gives an extension of our algorithms to their formulation of the problem. It is easy to do *same-set*(x, y) in the sequential setting: find the root u of the tree containing x, find the root v of the tree containing y, and test whether $u = v$. As Anderson and Woll observed, this does not suffice in the concurrent setting, because u might no longer be a root when the equality test occurs, possibly invalidating the test. Their solution has three cases. If $u = v$, return true: x and y are in the same tree when the test occurs, and remain in the same tree. If $u \neq v$, test whether u is still a root. If so, return false: x and y were in different trees when v was computed, since u and v were different roots. If not, redo the computation: do new finds from u and v, and repeat the test or tests. The following pseudocode implements this method:

Algorithm 16 : Concurrent same-set algorithm.

```
1: procedure same-set(x, y)
2:     u ← find(x); v ← find(y)*
3:     while u ≠ v do
4:         w ← u.p
5:         if u = w then return false
6:         u ← find(u); v ← find(v)*
7:     return true
```

The linearization point of a same-set is the last assignment to v. All our analyses of find and unite extend to include *same-set* as an allowed operation.

7.5 Concurrent Linking by Rank

To obtain a good work bound, we combine one-try or two-try splitting with a good linking method. Linking by random index is one such method, but our analysis of it assumes that the scheduling of CAS instructions is independent of the random node order. This assumption is questionable; if it fails, the work bound becomes much worse as a function of p, as we show in Section 7.9. To overcome this, we develop two concurrent versions of linking by rank, one deterministic and one randomized, both of which have good work bounds. To simplify our descriptions, we assume for the moment that the rank and parent of a node can be stored in a single block of memory that is updatable by one CAS instruction. In Section 7.6 we show how to eliminate this assumption.

Both of our versions of linking by rank are refinements of a generic method. The generic method links roots of different ranks using CAS, and links roots of the same rank using method *elink*, to be defined. The rank of node u is $u.r$, initially zero. The following pseudocode implements the generic method:

Algorithm 17 : Concurrent linking by rank algorithm.

```
1: procedure link(u, v)
2:     r ← u.r; s ← v.r
3:     if r < s then CAS((u.p, u.r), (u, r), (v, r))*
4:     else if r > s then CAS((v.p, v.r), (v, s), (u, s))*
5:     else elink(u, v, r)*
```

Given two roots u and v with ranks r and s, respectively, this method compares r to s. If $r < s$, it uses a CAS to make v the parent of u while guaranteeing that neither the parent nor the rank of u changes in the meantime. If $r > s$, it proceeds symmetrically. If $r = s$, it does an *elink* to link u and v. A link is successful if its CAS returns true or its *elink* is successful, in which case the linearization point of the link is its CAS or that of its *elink*. Our two versions of linking by rank differ only in their implementation of *elink*.

7.5.1 Linking by Rank via DCAS

A simple way to do *elink*(u, v, r) is to use a DCAS to make v the parent of u and increment the rank of v while guaranteeing that the ranks and parents of u and v do not change in the meantime.. The following pseudocode implements this idea:

Algorithm 18 : Concurrent linking by DCAS algorithm.

```
1: procedure elink(u, v, r)
2:     DCAS((u.p, u.r), (u, r), (v, r), (v.p, v.r), (v, r), (v, r + 1))*
```

An elink is successful if its DCAS returns true, in which case the linearization point of the elink is its DCAS.

Our first version of linking by rank uses this implementation of elink. The rank of a node can never decrease, and can increase only while the node is a root. It follows that the rank of a child is always strictly less than that of its parent. Linking by rank is an implicit form of linking by index: the successful links respect any total order consistent with the final ranks of nodes. Thus Theorem 7.4.1 holds for this method.

The following lemma and theorem extend known bounds on sequential linking by rank [196] to linking by rank via DCAS:

Lemma 7.5.1. *With linking by rank via DCAS, the sum of ranks is at most $n-1$, the number of nodes of rank k is at most $n/2^k$, and the maximum rank and the height of the union forest are at most* $\lg n$.

Proof. For a node to increase in rank by 1, it must be a root, and another root must become its child at the same time. It follows that the number of rank increments, and hence the sum of ranks, is at most $n - 1$, one per root that becomes a child. An induction on k shows that at most $n/2^k$ nodes can ever attain rank k. The bounds on the maximum rank and the height of the union forest follow, since no node can have rank exceeding $\lg n$. □

Theorem 7.5.2. *Linking by rank via DCAS in combination with any valid compaction method maintains the invariant that the parents define a set of trees that partition the nodes into the correct disjoint sets, and the rank of a child is less than that of its parent. Furthermore each set operation stops in $O(\log n)$ steps, so the algorithm is bounded wait-free.*

Proof. The first half of the theorem follows by induction on the number of steps as in the proof of the first half of Theorem 7.4.1. A find takes $O(\log n)$ steps by the argument in the proof of Theorem 7.4.1, since the height of the union forest is $O(\log n)$ by Lemma 7.5.1. We prove the bound for unites by an extension of the argument in the proof of Theorem 7.4.1. The nodes visited during a unite are on two paths in the union forest, and on each path they are visited in increasing order by rank. Each node visit takes O(1) steps, but roots can be visited many times. We charge

each repeated visit to a root either to a root becoming a child or to a root increasing in rank. Consider the nodes u and v just before an execution of the test "$u \neq v$" in unite. Each of u and v was a root at some time during the find that computed it. Suppose the test "$u \neq v$" succeeds. The next execution of link sets r to the rank of u and s to the rank of v. If $r < s$, then after the CAS either the rank of u has increased or u has become a child, whether or not the CAS succeeds. We charge the next visits to u and v to the rank increase of u or to u becoming a child. The symmetric argument applies if $r > s$. If $r = s$, at least one of u and v has increased in rank or become a child after the elink. We charge the next visits to u and v to whichever of these events has occurred. There are at most $2 \lg n$ roots that become children and at most $2 \lg n$ rank increases by Lemma 7.5.1, since for each of the two paths in the union forest the rank increases sum to at most $\lg n$. It follows that the total number of node visits during the unite, and hence the total number of steps, is $O(\log n)$. □

The efficiency of this linking method (though not its correctness) depends critically on using CAS to link nodes of different ranks, reserving DCAS for the equal-rank case. An attempted link of equal-rank nodes u and v using DCAS fails only if some other process makes u or v a non-root, or increases the rank of u or v. In the proof of Theorem 7.5.2 we charge extra node visits resulting from the failure of the DCAS to whichever of these events occurs. If we were to use DCAS to try to make a node v the parent of a node u of lower rank, the DCAS could fail because another process made v the parent of another node w. This changes neither the parent nor rank of u, nor of v, leaving us with no event to charge for extra node visits. In the worst case, $O(n)$ such links could produce $\Omega(pn)$ failures, resulting in total work linear in p. Using CAS to link nodes of different ranks eliminates these failures. Although we can avoid such interference in the disjoint set union problem as we have defined it, this is much harder to do in some extensions of the problem, as we discuss in Section 7.10.

7.5.2 Randomized Linking by Rank

To link equal-rank nodes using CAS, we need to do the parent change and the rank increment separately. The question is which one to do first. Making this decision randomly gives an approximation to linking by rank that produces few enough rank ties that we are able to get good work bounds. Since this method allows rank ties, we use linking by index to break such ties, in order to prevent the creation of non-trivial cycles of parent pointers. Assume that "$<$" is an arbitrary

total order of the nodes. To link two equal-rank roots u and v such that $u < v$, we flip a fair coin. If it comes up heads, we attempt to make v the parent of u; if it comes up tails, we attempt to increase the rank of u. The following pseudocode implements this idea. Random Boolean method *flip* returns true with probability $1/2$ and false otherwise, independent of all other flips.

Algorithm 19 : Concurrent randomized linking by rank algorithm.

```
1: procedure elink(u, v, r)
2:     if u < v then
3:         if flip then CAS((u.p, u.r), (u, r), (v, r))*
4:         else CAS((u.p, u.r), (u, r), (u, r + 1))
5:     else
6:         if flip then CAS((v.p, v.r), (v, r), (u, r))*
7:         else CAS((v.p, v.r), (v, r), (v, r + 1))
```

An elink is successful if it does a CAS that changes a parent pointer, in which case the linearization point of the elink is its CAS.

Our second version of linking by rank uses this implementation of elink. Observe that the CAS done after a flip is almost the same whether the flip returns true or false, the only difference being the updated field (parent or rank, respectively). In our analysis we shall assume that the success or failure of the CAS following a flip is independent of the outcome of the flip. In Section 7.6 we describe how to modify the implementation to eliminate the need for this independence assumption. Randomized linking by rank is an implicit form of linking by index: links respect the total order defined by final node ranks with ties broken by "$<$" .

Lemma 7.5.3. *With randomized linking by rank,* (i) *any node x has $O(1)$ ancestors of the same rank, in expectation;* (ii) *the sum of ranks is at most n in expectation and $n+O(n^{1/2})$ with probability $1-1/n^c$ for any constant $c > 0$, where the constant factor in the "O" depends on c;* (iii) *the expected number of nodes of rank at least k is at most $n/2^k$, and with probability at least $1 - n/2^k$, all nodes have rank less than k;* (iv) *the maximum rank is at most $\lg n + 3$ in expectation and is at most $(c + 1) \lg n$ with probability at least $1 - 1/n^c$ for any positive constant c;* (v) *the depth of the union forest is at most $3 \lg n + 9$ in expectation and $O(\log n)$ with probability at least $1 - 1/n^c$ for any constant $c > 0$, where the constant factor inside the "O" depends on c; and* (vi) *for a large enough constant c and any $k > 0$, the expected number of nodes of rank less than k and height at least ck in the union forest is at most $n/2^k$.*

Proof. Consider only flips that result in successful CAS operations. Each such flip produces a rank

increment with probability $\frac{1}{2}$; otherwise, it makes a root into a child.

(i) The probability that a node has k ancestors of the same rank is at most $1/2^{k-1}$. Summing gives the bound.

(ii) There are at most $n - 1$ flips that make a root into a child. The sum of ranks, which is the number of rank increments, is thus at most the number of heads in a sequence of coin flips containing at most n tails, which is at most n in expectation, and at most $n + O(n^{1/2})$ with probability $1 - n^c$ for any constant $c > 0$ by a Chernoff bound [35], with the constant factor in the "O" depending on c.

(iii) The rank of a given node is at least k with probability at most $1/2^k$. The expected number of nodes of rank at least k is thus at most $n/2^k$. By a union bound, all nodes have rank less than k with probability at least $1 - n/2^k$.

(iv) For $c > 0$, the probability that the maximum rank is at least $\lg n + c$ is at most $n/2^{\lg n + c} = 1/2^c$. It follows that the expected maximum rank is at most $\lg n + \sum_{i=1}^{\infty} i/2^i \lg n + 2 \lg n + 3$, and the probability that the maximum rank exceeds $(c + 1)\lg n$ is at most $1/2^{c \lg n} = 1/n^c$.

(v) A node gains a proper ancestor of the same rank with probability at most $1/2$. Thus the expected number of proper ancestors of the same rank as that of a given node is at most $\sum_{i=1}^{\infty} i/2^i = 2$, which implies by (ii) that the expected depth of the union forest is at most $3\lg n + 9$. Let $c > 0$. By (ii) the maximum node rank is at most $(c + 3)\lg n$ with probability at least $1 - 1/n^{c+2}$. For any node, the probability that it has at least $(b + c + 3)\lg n$ proper ancestors in the union forest is at most the probability that a sequence of fair coin flips containing at most $(c + 3) \lg n$ heads contains at least $b\lg n$ tails. By a Chernoff bound, for b sufficiently large, this probability is at most $1 - 1/n^{c+2}$. The probability that at least one of the n nodes has more than $(b + c + 3)\lg n$ proper ancestors in the union forest is at most $2n/n^{c+2} \leq 1/n^c$.

(vi) Let x be any node. We claim that for some $c > 0$ the probability that x has an ancestor y of rank less than k such that the path from x to y contains ck edges is at most $1/2^k$. Part (vi) follows from the claim by a union bound. To prove the claim, consider the edges on the path from x to y in the union forest. Call such an edge *good* if its ends have different ranks and *bad* otherwise. Each edge has probability at least $1/2$ of being good, independent of the status of all other edges. The claim follows by a Chernoff bound. □

Theorem 7.5.4. *Randomized linking by rank in combination with any valid compaction method maintains the invariant that the parents define a set of trees that partition the nodes into the correct*

disjoint sets. The parent of any non-root node has rank no less than that of the node, and if the ranks are equal, the parent has larger index. Each set operation stops in $O(\log n)$ steps with probability $1-1/n^c$ for any $c > 0$, where the constant factor in the "O" depends on c. The algorithm is bounded wait-free.

Proof. Except for the fact that the algorithm is bounded wait-free, the theorem follows from parts (iv) and (v) of Lemma 7.5.3 by a proof like those of Theorems 7.4.1 and 7.5.2.

To prove that the algorithm is bounded wait-free, we observe that for a node to increase in rank some larger node must have the same rank. It follows by induction that the i^{th} largest node has rank at most $i-1$, so the maximum rank is at most $n-1$. □

7.5.3 Linking by Random Index

With an appropriate definition of rank, Lemma 7.5.3 and Theorem 7.5.4 hold for linking by random index, under a strong independence assumption. We define the rank of node x to be $\lg n - \lg(n - x + 1)$. Thus node n has rank $\lg n$, nodes $n-1$ and $n-2$ have rank $\lg n - 1$, and so on. The rank of a child is no greater than that of its parent. We use these ranks only in the analysis; the implementation of the algorithm does not use them.

We assume that the random node order is independent of the linearization of the unite operations. More precisely, we assume that the node order and linearization are generated together in the following way. The implementation maintains a set U of unordered pairs $\{u, v\}$ that are candidates for linking, initially empty, and a partial order P of the nodes, initially empty, that is a total order on the nodes of any set defined by the links done so far, and that leaves any two nodes in different sets unordered. To do *link*(u, v), a process adds the unordered pair $\{u, v\}$ to U.

The scheduler sequentially removes pairs from U in arbitrary order. When removing a pair $\{u, v\}$ from U, the scheduler performs three actions. First, it modifies P by merging the total orders of the sets containing u and v, with each possible merged order equally likely. Second, if $u < v$ it sets $u.p = v$, if $v < u$ it sets $v.p = u$. This unites the sets containing u and v. The link corresponding to the pair $\{u, v\}$ succeeds. Third, the scheduler deletes from U all other pairs containing u or v. Each link corresponding to such a pair fails. When a pair is deleted from U, the process that added the pair to U proceeds with its next operations, which are the recomputing of its u and v.

The updating of P maintains the invariant that the total order of the nodes in any set defined by the links done so far is uniformly random. If there is more than one set after all unites have

been done, we can extend the final partial order to a total order by merging the total orders on the final sets, with each possible merged order equally likely. The result is a uniformly random permutation of the nodes, equivalent to a uniformly random numbering of the nodes from 1 to n. Thus this implementation does linking by random index, subject to the restriction imposed on the scheduler. The execution does not change if we initially number the nodes uniformly at random but reveal to the scheduler only the total order within each set formed so far.

This implementation restricts the behavior of linking by random index in at least two different ways: in the actual implementation, the CAS operation for a link is defined by an ordered pair, not an unordered pair, so the scheduler gets information about the node order before it needs to decide which such CAS to do next. Also, in the actual implementation a link can make a root the child of a non-root, which cannot happen with the scheduler constraint. We think the latter restriction is inconsequential, but the former is significant. We thus view our analysis of linking by random index as suggestive, not definitive.

Lemma 7.5.5. *With linking by random index, if the scheduler restriction holds, then parts* (i)*,* (v)*, and* (vi) *of Lemma 7.5.3 are true, as well as the following strengthened versions of parts* (ii)*,* (iii)*, and* (iv)*:* (ii) *the sum of ranks is at most* n*,* (iii) *the number of nodes of rank at least* k *is at most* $n/2^k$*, and* (iv) *the maximum node rank is at most* $\lg n$*.*

Proof. Parts (ii), (iii), and (iv) are immediate from the definition of ranks. Parts (i), (v), and (vi) follow as in the proof of Lemma 7.5.3 from the following claim: □

(*) Given a node x, each successive ancestor of x in the union forest has probability at least ½ of having higher rank then its parent, independent for each ancestor.

To prove (*), consider a node x. Given an execution of the algorithm, we modify the linearization by delaying links uniting sets not containing x until such a set is a subset of a set about to be united with x. That is, let S be the current set containing x, let $\{u, v\}$ be the next pair deleted from U with u but not v in S, and let S' be the current set containing v. We modify the schedule to delay all the links forming S' until just before the link of u and v. This does not change the steps done by the execution, only their linearization order. We generate the partial order P by generating a numbering of the nodes incrementally and revealing to the scheduler only the total order within each set constructed so far. Initially we assign a number uniformly at random to x. Subsequently when the scheduler removes a pair $\{u, v\}$ from U, if u and/or v is unnumbered we assign it a number chosen uniformly at random from the numbers not yet assigned.

Suppose the scheduler removes a pair $\{u, v\}$ from P with u the root of the tree containing x, and v in another tree, and that the corresponding link succeeds. Let S and S', respectively, be the sets containing u and v just before $\{u, v\}$ is removed from U. Just before the links forming S' are done, the only numbered nodes are those in S, and u has the largest number. Among the numbers larger than that of u, at least half have rank larger than that of u. When S' is formed, its nodes are numbered uniformly at random from among the unassigned numbers. Given that a number assigned to a node in S' is larger than that of u, it has probability at least $1/2$ of being larger than the rank of u. Since v has largest number among the nodes in S', if the link makes v the parent of u the rank of v is greater than that of u with probability at least $1/2$. The claim (*) follows by induction on the number of steps.

Theorem 7.5.6. *Theorem 7.5.4 holds for linking by random index if the scheduler restriction holds.*

Proof. The theorem follows from parts (iv) and (v) of Lemma 7.5.5 in the same way that Theorem 7.5.4 follows from parts (iv) and (v) of Lemma 7.5.3. □

7.6 Indirection and Helping

The algorithms in Sections 5.1 and 5.2 require that CAS (and DCAS in 5.1) support testing and updating of storage blocks able to store both the parent and the rank of a node. In this section we present two ways to eliminate the need for blocks to contain multiple fields. Both methods increase the number of steps per operation, but by at most a constant factor. We also discuss how to modify the implementation of the randomized linking algorithm of Section 7.5.2 to eliminate the need for an unrealistic independence assumption in its analysis.

The first method to reduce the block size, proposed by Anderson and Woll [12] is to use *indirection*. Specifically, each node contains only one field, a pointer to a *ledger* that contains the parent and rank of the node. To do a link via a CAS, a process creates a new ledger containing the updated information for the node being linked and then uses a CAS to attempt to replace the old ledger of the node by the new one. A link via a DCAS is similar, except that the process creates two new ledgers and uses a DCAS to replace the old ledgers of the two affected nodes. Parent updates done by splitting are done directly on the appropriate ledgers, without allocating new ones. The ledger method requires a way to allocate ledgers, and care must be taken to avoid the reuse of ledgers. The algorithm of Section 7.5.1 needs at most $3n - 2 + 2p = O(n)$ ledgers, one per initial set plus at most two per successful link plus at most two per process. The algorithm of Section 7.5.2 needs

$O(n)$ ledgers with high probability. If ledgers are used to implement randomized linking by rank, the independence assumption needed by the analysis becomes much weaker and quite realistic: the success or failure of a CAS can depend on all inputs to the CAS, in particular the ledger addresses, but not on the contents of the ledgers.

Allocating ledgers efficiently is itself a challenging problem, which Anderson and Woll ignored. One way to do it is to use the concurrent fetch and increment method of Ellen and Woelfel [57]. If ledgers are allocated individually, the number of steps to allocate a ledger is $O(\log p)$. If ledgers are allocated in groups of $O(\log p)$, the amortized time per allocation is $O(1)$ and the number of ledgers used will be $O(n)$ provided $p = O(n/\log n)$. If we are willing to use $O(n + p^2)$ memory, hazard pointers [154] and related techniques [1] can be used.

The second method is *helping*, as described for example in [87]. The idea is to allow processes to complete the tasks of other processes. We number the processes from 1 to p. Each node u has an extra field, *u.process*, which can hold a process number or 0, and is initially 0. Each process has a *descriptor* in which it records a sequence of steps it wants to perform. To update a node, a process writes appropriate instructions into its descriptor and then does a CAS or DCAS to write its process number into the *process* field of the affected node or nodes. Any other process that wants to update a node containing a non-zero process number must first execute the instructions in the corresponding descriptor. When the last instruction is executed, the process number in the affected node or nodes is reset to 0, allowing further updates to the node or nodes. As long as the number of instructions needed to do an update is bounded by a constant, the use of descriptors increases the total work by only a constant factor.

In using helping to link nodes of equal rank, we have to solve the *ABA problem*: A helping process, having completed the instructions in a descriptor, resets the process number in the relevant node to zero, but in the meantime the process being helped has reset the process number to zero, initiated a new update, and set the process number to its own number again. The helping process has no way to detect that a new update has been initiated by the process that initiated the old one. In our application, we can solve the ABA problem by using the monotonicity of ranks. This requires that a CAS be able to update the rank and the process number of a node as an atomic operation. Since ranks are small and in any realistic application $p \ll n$, we think this is a reasonable assumption.

The deterministic algorithm of Section 7.5.1 does links using helping as follows. Each descriptor contains two nodes and a rank. To link root x of rank r to root y, a process, say process i, writes x,

y, and r into its descriptor. If y has rank greater than r, it uses a CAS to write i into node x while verifying that the process number of x is 0 and the rank of x is r. If y has rank r, it uses a DCAS to write i into both x and y while verifying that the process numbers of x and y are 0 and the ranks of x and y are r. A process wanting to update a node that finds a non-zero process number i in the node reads the corresponding descriptor. Suppose the descriptor contains nodes x and y and rank r. The process sets $z = y.p$ and does a CAS to set the parent of x to z while verifying that x was a root before the update. It then tests whether y is a root of rank r. If so, it does a CAS to change the rank of y to $r + 1$ and the process number of y to 0 while verifying that the rank and process number of y were r and i before the update.

This method uses a couple of optimizations. It does not reset the process number of a node that becomes a non-root, since no subsequent link will try to change its parent or rank. Instead of making y the parent of x, it makes $y.p$ the parent of x. The reason to do the link this way is that some other process can make y a non-root just before process i does its CAS or DCAS to write i into x, or into x and y. If this happens, $y.p$ will have rank greater than r when x becomes its child, preserving the invariant that ranks strictly increase from child to parent. If this does not happen and the rank of y is r, the helping process adds one to the rank of y and resets the process number of y to 0.

The randomized algorithm of Section 7.5.2 does helping using descriptors containing a node y, a rank r, and a *flag* whose value is null, true, or false. A flag of true indicates that y should become the parent of the root containing the process number of the descriptor; a flag of false indicates that the rank of this node should be changed from r to $r + 1$. If process i wants to link root x of rank r to root y, it writes y and r into its descriptor. If y has rank greater than r, it sets the flag to true; if the rank of y equals r, it flips a fair coin and sets the flag correspondingly. Then it uses a CAS to set the process number of x equal to i while verifying that the rank and process number of x were r and 0 before the update. A process wanting to update a node x that finds a non-zero process number i in x reads the corresponding descriptor. If the flag is true it does a CAS to set the parent of x to y while verifying that x was a root before the update. If the flag is false it does a CAS to set the rank and process number of x to $r + 1$ and 0 while verifying that they were r and i before the update.

The analysis of this method relies on the same independence assumption as the method using ledgers: scheduling decisions are independent of the contents of descriptors. A variant of the method is to set the flag *after* the process number of the descriptor is written into x: the first step of a

helping process is to change the flag from null to true or false using the randomized CAS operation mentioned in Section 7.2. If this operation is available, no independence assumption is needed. In the next section, we give pseudo-code for an implementation using randomized CAS, alonging with a line-by-line explanation of the implementation. This implementation of randomized linking by rank satisfies the Anderson-Woll requirement that a randomized algorithm be efficient even if the scheduler knows the outcome of previous random choices. We think, though, that it is reasonable to assume that the scheduler makes its decisions only on the basis of the inputs to the CAS operations, or that it cannot read the private memories of the processes. If either of these assumptions hold, we do not need randomized CAS.

7.7 Our Algorithm with Randomized Compare-and-Swap

There are several ways of implementing randomized linking by rank using randomized CAS. We present what we think is the clearest and most concise implementation below. Our implementation uses a compressed tree structure, just as do the other sequential and concurrent algorithms we have presented, but with a small modification discussed below. The forest contains one rooted tree per set, whose nodes are the elements of the set and whose root is the set leader. We assume that the nodes have indices 1 through n and thus can be compared via '$<$'. Each node x has a field $x.p$ to store the address of a *parent* node, a field $x.r$ to store a non-zero integer *rank*, and a field $x.b$ to store a single *root-bit* signifying whether or not x is the root of its tree. Initially, $x.p = x$, $x.r = 0$, and $x.b = 1$. The rank is never more than n, and thus needs only $\lceil \log n \rceil$ bits of storage. We shall assume that all three fields of a node are stored in a single word in memory, as a triple $x.f = [x.p, x.r, x.b]$. The memory words are of size $2\lceil \log n \rceil + 1 = O(\log n)$ in this representation. In fact, with high probability the maximum rank is $O(\log n)$, and each node requires only $\log n + O(\log \log n)$ bits.

A note about our representation: If the root-bit $x.b$ is set to 1, x is a root, in which case our implementation ignores the value of the parent field $x.p$. In this way our representation differs from the classic representation, in which the parent field $x.p = x$ for a root node x. This fact is crucial to understanding the pseudo-code.

In the pseudo-code, \$ represents a random bit, i.e. a value with $Bernoulli(1/2)$ distribution.

Algorithm 20 is the pseudo-code for unite and link. The implementation of the *find*(x) procedure used in the code is discussed in the next subsection. To do *unite*(x, y), we start as in the sequential case by finding the roots u and v of the trees containing x and y, respectively (Line 2). If $u = v$,

then x and y are already in the same set and no work needs to be done (Line 3). Otherwise, x and y are in different trees, so we can try to link u and v by doing a CAS to make v the parent of u, or vice-versa (Line 4). We mark the *link* on Line 4 and subsequent linearization points with an asterisk in the code. Further explanation of the pivotal *link* procedure is in the next paragraph. But we must allow for the possibility of the CAS failing, which can happen for example if it tries to make v the parent of u but in the meantime some other process makes another node the parent of u. Notably, the CAS fails if the other process does exactly the same CAS and makes v the parent of u. A solution that works in either case is to simply continue walking up the tree from the present u and v (Line 5), until the paths intersect or another attempt at linking is necessary. This method was first proposed by Anderson and Woll [12] and was subsequently used by Jayanti and Tarjan [118].

Algorithm 20 : Pseudo-code to unite x and y.

```
1:  procedure unite(x, y)
2:      u ← find(x); v ← find(y)*
3:      while u ≠ v do
4:          link(u, v)*
5:          u ← find(u); v ← find(v)*

6:  procedure link(u, v)
7:      [u_p, r, u_b] ← u.f
8:      [v_p, s, v_b] ← v.f
9:      if r < s then CAS(u.f, [u_p, r, 1], [v, r, 0])*
10:     else if s < r then CAS(v.f, [v_p, s, 1], [u, s, 0])*
11:     else
12:         if u < v then CAS(u.f, [u_p, r, 1], [v, r + 1, $])*
13:         else CAS(v.f, [v_p, s, 1], [u, s + 1, $])*
```

An explanation of our Linking heuristic: Link initially reads the fields of u and v (Lines 7-8). If r, the rank of u, is less than s, the rank of v, the link attempts to change the parent of u to v and the root-bit of u to 0 ("not a root"), while leaving the rank of u unchanged (Line 9). If s is less than r, the link proceeds symmetrically (Line 10). The interesting case is $r = s$: if $u < v$ (check on Line 12 succeeds), the algorithm tries to change the parent of u to v *and* increment the rank of u, *and* set the root-bit of u *randomly* (Line 12). This case deviates from our presentation in Algorithm 19, so we now explain why the algorithm tries to change *both* the parent and rank fields. If the update succeeds and the root-bit, $u.b$, gets set to 1, then u continues to be a root, and thus the parent field is disregarded by the algorithm, so it does not matter that it was changed to

v. On the other hand, if the update succeeds and root-bit gets set to 0, then u is no longer a root, and the rank field is subsequently disregarded by the algorithm, so it does not matter that it was changed to $r+1$. Therefore, although syntactically the algorithm changes both the parent *and* the rank fields, semantically only the parent *or* the rank changes. Thus the implementation matches the idea in Algorithm 19. Line 13 acts symmetrically in the case that $u.r = v.r$ and $v < u$.

Note: for the purpose of the analysis only, we think of the rank as *not* incremented if Line 12 or 13 makes a root a non-root.

7.7.1 The Find Procedure

We present the two different implementations of *find*(x), namely *naïve* and *two-try splitting* in Algorithm 21. These procedures reproduce Algorithm 13 and Algorithm 15 using the node representation with three fields (parent, rank, and root-bit) per node.

An explanation of Naïve find: Naïve find uses an auxiliary variable u (Line 2) that follows parent pointers up the tree (Line 4) until it reaches a root (Line 3). *find*(x) returns this node as the leader (Line 5). The linearization point of this procedure is the time at which u is discovered to be a root.

Algorithm 21 : Find algorithms naïve and two-try splitting, respectively

```
1:  procedure find(x)
2:      u ← x
3:      while not u.b* do
4:          u ← u.p
5:      return u

6:  procedure find(x)
7:      u ← x;
8:      while not u.b* do
9:          [v, r, u_b] ← u.f; [w, s, v_b] ← v.f*
10:         if v_b then return v
11:         CAS(u.f, [v, r, 0], [w, r, 0])
12:         [v, r, u_b] ← u.f; [w, s, v_b] ← v.f*
13:         if v_b then return v
14:         CAS(u.f, [v, r, 0], [w, r, 0])
15:         u ← v
16:     return u
```

An explanation of the two-try splitting find pseudo-code: Find uses an auxiliary variable u to walk up the tree (Line 7). If u is not a root (Line 8), its parent v, and grandparent $v.p = w$

are read (Line 9). If v is a root, Find has succeeded and can return v (Line 10); otherwise, an improvement is attempted on Line 11. A successful CAS on Line 11 changes only the parent of u, without modifying the other fields of u. After a second attempt to improve the same node u (Lines 12-14), u is replaced by its parent v, in order to keep walking up the tree (Line 15). Finally, if u becomes the root (Line 8), it is returned on Line 16. The linearization point of the procedure is the time when the root-bit of the returned node is read, since this is the time when the returned node is surely the root of the tree.

Note: Removing the second attempt to improve u (Lines 12-14) from the pseudo-code of find with two-try splitting produces pseudo-code for find with one-try splitting.

7.8 Upper Bounds

The results of Sections 7.5 and 7.6 give us the following theorem:

Theorem 7.8.1. *With any of the three linking methods of Section 7.5 combined with any valid compaction method, the total work is $O(m \log n)$. This bound is worst-case for the deterministic linking method, high-probability for the randomized methods. If randomized linking by rank is implemented as described in Section 7.6, the bound is valid even for an adversarial scheduler.*

Proof. The theorem is immediate from the results of Sections 5 and 6. □

The use of splitting instead of naïve find improves the total work bounds significantly if $p \ll n$. We show this by extending the analysis of sequential splitting [196, 75] to one-try and two-try splitting.

We define the *density* d of a set union problem instance to be $m/(np)$ if splitting is two-try, $m/(np^2)$ if splitting is one-try. We shall obtain a bound of $O(m \cdot (\alpha(n, d) + \log(1 + 1/d)))$ on the total work if either kind of splitting is used in combination with any of the three linking methods. The main obstacle we encounter in extending the sequential analysis to the concurrent setting is accounting for unsuccessful CAS operations. Accounting for such operations adds the logarithmic term to the work bound.

We call a problem instance *sparse* if $d < 1$ and *dense* otherwise. The logarithmic term in the work bound dominates only in sparse instances. We start with the analysis of dense instances, which is simpler than that of sparse ones.

We call a child a *zero child* if its rank is the same as that of its parent. Zero children only exist if a randomized linking method is used.

With deterministic linking by rank or linking by random index, ranks are at most is $\lg n$. With randomized linking by rank, they are at most $n-1$, although large ranks occur with exponentially small probability (Lemma 7.5.3 part (iii)).

7.8.1 The Dense Case

Throughout this section we assume $d \geq 1$.

Lemma 7.8.2. *The number of finds is $O(m)$, worst-case unless randomized linking by rank is used, in which case the bound is with high probability.*

Proof. There are at most two finds per unite plus at most two per process per root that increases in rank or becomes a child, for a total of $O(m + np) = O(m)$. For randomized linking, this bound follows from part (ii) of Lemmas 7.5.3 and 7.5.5 and is high-probability for randomized linking by rank, worst case for linking by random index. □

We call a node *low* if its rank is less than d and *high* otherwise. All nodes have rank at most $n-1$. During a find, a *visit* to a node is an iteration of the find loop in which the node is the value of u. (See the pseudocode in Section 7.4.)

Lemma 7.8.3. *The number of visits to low nodes during finds is $O(m)$, worst-case if linking is deterministic, expected if randomized.*

Proof. Consider three successive visits to low nodes during a find, to u, v, and w. Let I be the interval of time between the visits of u and w. We claim that at least one of the following events occurs during I: u or v becomes a child, $u.p.r$ increases, or u or v loses an ancestor of the same rank. The number of such events for fixed u and v is $O(d)$: a node only becomes a child once, its parental rank can increase at most d times before it exceeds d and its parent is not low; a node has $O(1)$ ancestors of the same rank in expectation by part (i) of Lemmas 7.5.3 and 7.5.5. We charge the visit to u to the corresponding event (or any such event if there is more than one). Each event is charged for at most $2p$ visits, at most two per process. (The factor of two comes from the two nodes associated with a visit, the node itself and the next node visited.) Summing over all nodes, we obtain a bound of $O(npd) = O(m)$ on visits to low nodes.

Suppose the claim is false. Then u and v are children when u is visited. After the CAS following the visit to u, the parent of u has changed; after the CAS following the visit to v, the parent of v has changed. If either u or v is a zero child when u is visited, at least one of them becomes a non-zero child or loses an ancestor of the same rank during I. Thus neither u nor v is a zero child when u is visited. But then the rank of the parent of u increases by the time the CAS after the visit to u finishes, making the claim true. □

Bounding visits to high nodes is more complicated. For each high child x, we measure the progress of compaction by keeping track of an increasing function of the rank of the parent of x, called the *count* of x. We define counts using Ackermann's function. Our formulation is an extension of that of Kozen [132]. We define the *level* $x.a$ of a high node x, and the *index* $x.b$ and *count* $x.c$ of a high child x, as follows:

$x.a = \min\{k|\ A_k(x.r) > x.p.r\}$;

$x.b = \max\{i|\ A_{x.a}(i) \leq x.p.r\}$;

$x.c = x.r \cdot x.a + x.b$.

We bound the range of levels, indices, and counts by using the properties of Ackermann's function:

Lemma 7.8.4. *If x is a high node, $0 \leq x.a \leq \alpha(n,d)$ and $x.a = 0$ if and only if $x.r = x.p.r$. If x is a high child, $0 \leq x.b < x.r$ and $0 \leq x.c < (\alpha(n,d)+1)x.r$. The values of $x.a$ and $x.c$ never decrease, and if $x.a$ or $x.b$ increases, $x.c$ increases by at least as much.*

Proof. Since $A_0(x.r) = x.r + 1$, $x.a = 0$ if and only if $x.r = x.p.r$, and $x.a \geq 0$ if it is defined. If x is a high node, $A_{\alpha(n,d)}(x.r) \geq A_{\alpha(n,d)}(\lfloor d \rfloor) > n > x.p.r$. Thus $x.a$ is defined and is at most $\alpha(n,d)$. (Here for randomized linking by rank we use the assumption that all ranks are less than n.) Suppose x is a high child. If $x.a = 0$, $x.b = x.r - 1$ since $x.r \geq d \geq 1$. If $x.a > 0$, $A_{x.a}(0) = A_{x.a-1}(1) \leq A_{x.a-1}(x.r) \leq x.p.r$, so $x.b$ is defined. Since $A_{x.a}(x.r) > x.p.r$, $x.b < x.r$. The bounds on $x.c$ follow from those on $x.a$ and $x.b$. While x is a root, $x.a = 0$. Once x is a child, $x.r$ is constant and $x.p.r$ cannot decrease, so $x.a$ cannot decrease by Lemma 7.3.1. While $x.a$ is constant, $x.b$ cannot decrease for the same reason. If $x.a$ increases by one, $x.b$ can decrease by at most $x.r - 1$, resulting in an increase of at least one in $x.c$. If $x.a$ increases by more than k, $x.c$ increases by at least $(k-1)x.r + 1$. □

Lemma 7.8.5. *The sum of the counts of all high children is $O(n\alpha(n,d))$, worst-case unless linking is randomized by rank, in which case the bound is high-probability.*

Proof. By Lemma 7.8.4, the sum of the counts of high children is $O(\alpha(n, d))$ times the sum of the ranks of all nodes. By Lemmas 7.5.1 and 7.5.5, the sum of ranks is less than n for deterministic linking by rank and linking by random index. For randomized linking by rank, it is $O(n)$ with high probability by Lemma 7.5.3. □

The following lemma is the key to the analysis of splitting.

Lemma 7.8.6. *Consider a time t at which u is a high child whose parent v is also a (high) child. Let w the parent of v at time t, and let $u.a$, $v.a$, and $w.r$ be the levels of u and v and the rank of w at time t, respectively. Suppose that at time t or later the parent of u changes from v to a node x of rank at least $w.r$. If $v.a > u.a$, the parent change increases $u.a$ and $u.c$ by at least $v.a - u.a$; if $v.a = u.a$, the parent change increases $u.c$ by at least 1 or causes u to lose an ancestor of the same rank.*

Proof. Let $u.r$ and $v.r$ be the ranks of u and v at time t, respectively. Let $x.r$ be the rank of x when it becomes the parent of u. Since $A_{v.a-1}(u.r) < A_{v.a-1}(v.r) \leq w.r \leq x.r$, the level of u after the parent change is at least $v.a$. If $v.a > u.a$, the parent change increases the level and hence the count of x by at least $v.a - u.a$ by Lemma 7.8.4. Suppose $v.a = u.a$. If $u.a = 0$, the parent change causes u to lose v as an ancestor. Suppose $u.a > 0$. Since $A_{u.a}(u.b + 1) = A_{u.a-1}(A_{u.a}(u.i)) \leq A_{u.a-1}(v.r) \leq w.r \leq x.r$, the parent change increases either the level or the index of u and hence increases the count of u. □

To count visits to high nodes, we use a credit argument. One credit pays for one high-node visit. We allocate a certain number of credits to each find when it starts, and additional credits when high nodes increase in count or lose ancestors of the same rank. We show via a *credit invariant* that these credits suffice to pay for all the high-node visits. A bound on the total number of credits gives a bound on the number of high-node visits.

We begin by analyzing two-try splitting: even though it is more complicated than one-try splitting, its analysis is simpler. We call a find *active* while it is being executed. When a find starts, we allocate it $\alpha(n, d) + 1$ credits. When the count of a high child increases by k, we allocate $2k$ credits to each active find, for a total of at most $2pk$. When a high child loses an ancestor of the same rank, we allocate one credit to each active find, for a total of at most p.

Lemma 7.8.7. *With two-try splitting, the number of allocated credits is $O(m\alpha(n, d))$, worst-case if linking is deterministic, average-case if randomized.*

Proof. By Lemma 7.8.2, the number of credits allocated to finds when they start is $O(m\alpha(n,d))$. By Lemma 7.8.5, the number of credits allocated to finds as a result of increases in count is $O(np\alpha(n,d)) = O(m\alpha(n,d))$. By Lemmas 7.5.3 and 7.5.5, the expected number of credits allocated to finds as a result of nodes losing ancestors of the same rank is $O(np) = O(m)$. □

Lemma 7.8.8. *With two-try splitting, just after a high node u is visited by a find, the find has at least $u.a$ credits.*

Proof. We prove the lemma by induction on the number of high-node visits done by a find. When the find starts, it has $O(\alpha(n,d)+1$ credits. The first visit costs one, leaving $\alpha(n,d)$, which is enough to make the lemma true just after this visit. Suppose the lemma holds just after u is visited, and let v be the next node visited. We denote by unprimed and primed values their values just after the visit to u and just before the visit to v, respectively. The lemma holds after the visit to v provided that the find accrues at least $v.a' - u.a + 1$ credits between the visits to u and v. To show that this happens, we need the following crucial inequality, which follows from Lemma 7.8.6:

(*) $u.a' \geq v.a$

To prove (*), we refer to the implementation of two-try splitting. Let t be the first time $u.p = v$. Time t is after the visit to u, since $u.p$ changes between the first and second times that the find sets its variable v after the visit to u, as a result of the first CAS during the visit to u succeeding or failing. Let w be the parent of v at time t. Consider the change to $u.p$ resulting from the second CAS after the visit to u. This change satisfies the hypothesis of Lemma 7.8.5, since the new parent of u must have been the parent of v at time t or later. By Lemma 7.8.6, just after this change to $u.p$, the level of u is at least the level of v at time t. Since levels are non-decreasing, (*) holds.

Between the visits to u and v, the find accrues at least $2(u.a' - u.a + v.a' - v.a) = (v.a' - u.a) + (u.a' - u.a) + (v.a' - v.a) + (u.a' - v.a)$ credits as a result of level increases.. Each of the last three terms is non-negative, the last one by (*). Thus the find accrues at least $v.a' - u.a + 1$ credits between the visits, unless the levels of u and v are equal and unchanging between the visits. Suppose the levels of u and v are equal and unchanging between the visits. By Lemma 7.8.6, the find accrues at least one credit when the parent of u changes from v. □

Lemma 7.8.9. *With two-try splitting, the number of visits to high nodes is $O(m\alpha(n,d))$, worst-case if linking is deterministic, average-case if randomized.*

Proof. The lemma is immediate from Lemmas 7.8.7 and 7.8.8. □

Now we extend the analysis to one-try splitting. The proof of Lemma 7.8.8 fails for one-try splitting, because a CAS done by one process, say process 1, can fail as a result of a successful CAS done by another process, say process 2, that sets its value of v *before* process 1's most recent high-node visit. That is, time t in the proof of Lemma 7.8.6 can precede the visit. This invalidates the use of Lemma 7.8.6 in the proof.

To overcome this problem, we allocate additional credits to node count increases, and we allow active finds to shift some of their credits to the other active finds. Specifically, when a find starts, we allocate it $\alpha(n, d) + 1$ *normal* credits. When a high child loses an ancestor of the same rank, we allocate one normal credit to each active find. When the count of a high node increases by k, we allocate $2k$ normal credits and $2k(p-1)$ *extra* credits to each active find. When a CAS in a find succeeds, we shift a $1/(p-1)$ fraction of the find's extra credits to each other active find. Shifted extra credits become normal; that is, we shift a credit at most once.

Lemma 7.8.10. *With one-try splitting, the number of allocated credits is $O(m\alpha(n, d))$, worst-case if linking is deterministic, average-case if randomized.*

Proof. The bound holds for normal credits by the proof of Lemma 7.8.7. By Lemma 7.8.5, the number of extra credits allocated to finds as a result of increases in count is $O(np^2\alpha(n, d)) = O(m\alpha(n, d))$ since $d = m/(np^2)$. □

Lemma 7.8.11. *With one-try splitting, just after a high node u is visited by a find, the find has at least $u.a$ normal credits.*

Proof. The proof is an extension of that of Lemma 7.8.8. Consider a find, say find 1. The credits allocated to the find when it starts make the lemma true just after its first high-node visit. Suppose the lemma holds just after find 1 visits u, and let v be the next node it visits. We consider three cases. If the CAS during the visit of find 1 to u succeeds, the lemma holds just after the visit to v by an argument like that in the proof of Lemma 7.8.8. (This case does not use shifted credits.) Suppose this CAS fails, because a CAS done by another find, say find 2, changes $u.p$ from v to another value. Let t be the last time that find 2 set its variable v before its successful CAS. If t is after find 1 visits u, the lemma holds just after the visit to v by an argument like that in the proof of Lemma 7.8.8, again without the use of shifted credits.

The third, new case is if t precedes the visit of find 1 to u. Let t', t'', and t''' be the times find 1 visits u, find 2 does its CAS, and find 1 visits v, respectively. We denote by unprimed, primed, double-primed, and triple-primed values their values at times t, t', t'', and t''', respectively. Applying

Lemma 7.8.5 to time t and the successful CAS of find 2 gives $u.a'' \geq$ v.a; and, if $u.a \leq v.a$, the count of u increases by at least 1 or u loses an ancestor of the same rank when find 2 does its CAS.

At time t', find 1 has at least $u.a'$ normal credits by the induction hypothesis. Between times t' and t''', it accrues at least $2(u.a''' - u.a' + v.a''' - v.a')$ normal credits. Between times t and t'', find 2 accrues at least $2(p-1)(u.a'' - u.a + v.a'' - v.a) \geq 2(p-1)(u.a' - u.a + v.a' - v.a)$ extra credits, of which at least $2(u.a' - u.a + v.a' - v.a)$ are shifted to find 1 and become normal at time t'': find 1 is active at t'' since its CAS fails as a result of the CAS by find 2 succeeding. Thus between t' and t''' find 1 accrues at least $2(u.a''' - u.a + v.a''' - v.a) \geq (v.a''' - u.a') + (u.a''' - u.a) + (v.a''' - v.a) + (u.a'' - v.a)$ normal credits. Since $u.a'' \geq v.a$, this is at least $v.a''' - u.a' + 1$, enough to make the lemma true for the visit to v, unless u and v have equal and unchanging levels from t to t''', in which case find 1 accrues a normal credit when find 2 does its CAS. □

Lemma 7.8.12. *With one-try splitting, the number of visits to high nodes is $O(m\alpha(n, d))$, worst-case if linking is deterministic, average if randomized.*

Proof. The lemma is immediate from Lemmas 7.8.10 and 7.8.11. □

7.8.2 The Sparse Case

In this section we modify the analysis of Section 7.8.1 to handle sparse instances. Throughout this section we assume $d < 1$. We need to change the definition of low and high nodes, add an additional node type, *middle*, and (for the purpose of the analysis only) redefine the ranks of nodes.

Let $l = \lg(1 + 1/d)$. Since $d < 1$, $l > 1$. A node is *low* if its rank is less than l and its height is less than cl, where c is the constant in part (vi) of Lemmas 7.5.3 and 7.5.5; *middle* if its rank is less than l but its height is at least cl, and *high* if its rank is at least l. Middle nodes can exist only if linking is randomized.

Lemma 7.8.13. *The number of non-low nodes is at most $2nd$, as is the sum of the ranks of such nodes. This bound is worst-case if linking is deterministic, average-case if randomized.*

Proof. By part (vi) of Lemmas 7.5.3 and 7.5.5, the expected number of middle nodes is at most $n/2^l \leq n/2^{\lg(1/d)} = nd$ if linking is randomized. (It is zero if not.). By Lemma 7.5.1 or part (iii) of Lemma 7.5.3 or 7.5.5 depending on the linking method, the number of high nodes is also at most $n/2^l \leq nd$, worst-case if linking is deterministic or by randomized index, average-case if by randomized rank. The bound on the sum of ranks follows from the node bound by the argument in

the proof of Lemma 7.5.1 if linking is deterministic, by that in the proof of part (ii) of Lemma 7.5.3 or 7.5.5 if randomized. □

Lemma 7.8.14. *The number of finds that visit at least one non-low node is $O(m)$, worst-case if linking is deterministic, average-case if randomized.*

Proof. Consider the finds during unites that visit at least one non-low node. At most two per unite also visit a low node. Of those that visit only non-low nodes, there are at most two per unite plus at most $2p$ per non-low node that becomes a child or has a rank increase, two per process doing a unite while the event in question takes place. By Lemma 7.8.13, the number of such finds is $O(npd) = O(m)$. □

Lemma 7.8.15. *The number of visits to low nodes is $O(ml)$, worst-case.*

Proof. The analysis of node visits in the proof of Theorem 7.5.2 restricted to nodes of rank less than l and height less than cl gives a bound of $O(l)$ low-node visits for each find and unite. □

Lemma 7.8.16. *If linking is randomized, the expected number of visits to middle nodes is $O(ml)$.*

Proof. By Lemma 7.8.14, the expected number of finds that visit middle nodes is $O(m)$. During such a find, each visit to a middle node except the last two is followed by a middle node losing a child of the same rank or the parent of a middle node x increasing in rank. The latter can only happen l times before x has a parent that is not a middle node; subsequently, x can only be the last middle node visited during a find. We charge each visit to a middle node other than the last two of a find to the corresponding event. The charge per event is at most p, and the expected number of events is at most ndl rank increases and $O(nd)$ losses of same-rank ancestors, the latter by part (i) of Lemma 7.5.3 or 7.5.5. Such events account for $O(npdl) = O(ml)$ visits. Adding the last two per find gives the lemma. □

To count visits to high nodes, we define the *effective rank* of a high node x to be $x.er = x.r - l + 1$. We define levels of high nodes and indexes and counts of high children, using effective ranks in place of ranks. Since the effective rank of a high node is at least one, levels of high nodes and indices and counts of high children are well-defined. We allocate credits exactly as in Section 7.8.1. Lemmas 7.8.4 and 7.8.6 remain true. By Lemma 7.8.13, the sum of counts of high children is $O(nd\alpha(n, d))$, worst-case if linking is deterministic, high-probability if randomized. We allocate credits exactly as in Section 7.5.1. Lemmas 7.8.8 and 7.8.11 remain true. If splitting is two-try, the

number of allocated credits is $O((m+ndp)\alpha(n,d)) = O(m\alpha(n,d))$ since $d = m/(np)$; if splitting is one-try, it is $O((m+ndp^2)\alpha(n,d)) = O(m\alpha(n,d))$ since $d = m/(np^2)$. We conclude that Lemmas 7.8.7, 7.8.9, 7.8.10, and 7.8.12 hold in the sparse case (with the new definition of a high node).

7.8.3 The Total Work Bound

Combining the results of Sections 7.1 and 7.2, we obtain the following theorem:

Theorem 7.8.17. *With any of the three linking methods of Section 7.5 and either one-try or two-try splitting, the total work is $O(m(\alpha(n,d) + \log(1+1/d)))$, worst-case if linking is deterministic, average-case if randomized, where $d = m/(np^2)$ if splitting is one-try, $d = m/(np)$ if splitting is two-try.*

Proof. The theorem follows from Lemmas 7.8.3, 7.8.9, 7.8.12, 7.8.15, and 7.8.16. □

7.9 Lower Bounds

In this section, we derive lower bounds on the worst-case and amortized efficiency of set union algorithms. In the first subsection, we prove lower bounds on the work efficiency of the algorithms by explicitly providing worst-case executions—both the operations and the adversarial schedules. At a high level, our executions are constructed by the following observations and steps. For each algorithm, we describe operations that build a tree of logarithmic height using *unite* operations. We observe that *shadowing schedules* in which all processes are scheduled in lock-step while performing the same expensive *find* operations result in worst-case behavior. We apply a shadowing schedule to processes performing a *find* on the deepest node in the aforementioned tree to prove that the logarithmic term in our upper bounds is tight. Then, we combine the idea of shadowing schedules with previous sequential lower bounds of Tarjan et al. and Fredman et al. [196, 64] to show that the inverse-Ackermann term in our upper bounds is tight. Our algorithmic lower bounds section proves that our amortized upper bound analyses are tight when find operations are done with two-try splitting.

In the second subsection, we show general lower bounds that apply to the concurrent set union problem. First, we prove that, in the worst-case, any concurrent set-union algorithm must do at least $\Omega(\log \min\{n,p\})$ work in expectation for a single operation. When $p = n^{\omega(\frac{1}{\log\log n})}$, this lower bound is stronger than the sequential lower bound of $\Omega\left(\frac{\log n}{\log\log n}\right)$ given by Fredman and Saks [64]

in the cell probe model. It also shows a separation in work complexity between the sequential and concurrent versions of the set-union problem, since Blum [25] presented an algorithm that does at most $O\left(\frac{\log n}{\log \log n}\right)$ work per operation in the sequential setting. Furthermore, whenever $\log p = \Theta(\log n)$, i.e. when $p = n^{\epsilon}$, this lower bound establishes that randomized linking with any form of compaction yields an algorithm with optimal expected work per operation. Finally, we generalize the worst-case lower bound using shadowing schedules to show that our algorithm obtained by combining randomized linking with two-try splitting is optimal amongst a class of *symmetric algorithms* that includes all known algorithms for the concurrent disjoint set union problem.

7.9.1 Algorithmic Lower Bounds

In order to prove the tightness of the inverse-Ackermann term in our upper bounds, we recall a sequential cell probe lower bound on the set union problem given by Fredman and Saks.

Lemma 7.9.1 ([64])**.** *Let $\mathcal{A}$ be any randomized algorithm that solves the sequential set union problem. For any fixed number of nodes n, and any $M \geq n$, there is a sequence of operations σ_M, that makes $\mathcal{A}$ perform $\Omega(M\alpha(n, M/n))$ expected work.*

We use Lemma 7.9.1 to establish a concurrent lower bound.

Lemma 7.9.2. *Let $\mathcal{A}$ be any of the algorithms we have described for concurrent set-union. There is some sequence of m operations using p processes on n nodes that requires $\Omega\left(m \cdot \alpha\left(n, \frac{m}{np}\right)\right)$ work in expectation.*

Proof. Any concurrent algorithm is also a sequential algorithm if it is run by a single process. So, for any given $M \geq n$, we can take a worst-case sequence σ_M of operations from Lemma 7.9.1. That is, a single process running the sequence of operations σ_M will perform $\Omega(M\alpha(n, M/n))$ work in expectation. In the remainder of the proof, we use shadowing schedules, in which processes run in lock-step with each other and thereby do not gain locally from any compaction attempts of other processes, to get the lower bound.

We consider two cases for m:

Case 1: If $m \geq np$, then we choose $M = m/p \geq n$. If each of the p processes runs σ_M and is scheduled in lock-step (so that the processes all walk up find sequences together and do not benefit from each other's compaction attempts), then the total number of operations is $pM = m$ and the total amount of work is $\Omega(pM\alpha(n, M/n)) = \Omega\left(m\alpha\left(n, \frac{m}{np}\right)\right)$.

Case 2: If $m < np$, we choose $M = n$. Then, σ_M performed by a single process takes $\Omega(n\alpha(n,1))$ expected work. We observe that $m/n < p$ and assign m/n processes the operation sequence σ_M, thus assigning m operations. If the processes are scheduled in lock-step, the total expected work performed by them is $\Omega(m/n \cdot n\alpha(n,1)) = \Omega\left(m\alpha\left(n, \frac{m}{np}\right)\right)$.

□

We can also show that the logarithmic term $\log(\frac{np}{m} + 1)$ is an amortized lower bound for all our algorithms. The schedule that builds binomial trees with a single process and makes all the processes shadow each other up the longest branch of these trees yields the lower bound.

Lemma 7.9.3. *For randomized linking by rank and linking by DCAS, regardless of what type of compaction is used in find operations, and for any positive integer $k \in [1, n]$, there is a sequence of $k-1$ unite operations that will build a tree with k nodes with height $\Omega(\log k)$.*

Proof. For simplicity, we initially assume that k is a power of 2. Let B_j be the binomial tree of height j. All the nodes are initially in singleton trees, i.e. B_0 trees. We proceed in $\lg k$ rounds. In round r, we start with $\frac{k}{2^{r-1}}$ trees of type B_{r-1}, and simply unite their roots pairwise. After $\lg k$ rounds we end up with a single tree of type $B_{\lg k}$. If k were not a power of 2, we perform the above procedure with the largest power of 2 less than k and simply unite the remaining nodes to the root of the main tree. □

A slightly more complex construction allows us to prove a similar lemma for linking by index.

Lemma 7.9.4. *For randomized linking by index, regardless of what type of compaction is used in find operations, and for any positive integer $k \in [1, n]$, there is a sequence of $k-1$ unite operations by a single process that will build a tree with k nodes in which the depth of a uniformly randomly picked node is $\Omega(\log k)$ in expectation.*

Proof. The proof is constructive. The construction of these trees is inspired by *binomial trees*, and is done in multiple *rounds* such that each round fully finishes before the next round starts. Without loss of generality let k be a power of 2, as otherwise we could just use the greatest power of 2 less than k in the following construction. Initially, we let the nodes be in singleton trees $T_{1,1}, \ldots, T_{k,1}$. In each round we will combine pairs of trees, and each tree T will have a designated node $\nu(T)$. In the initial trees the designated node is the only node. In the first round we combine pairs of trees

by performing

$$unite(\nu(T_{1,1}), \nu(T_{2,1})), unite(\nu(T_{3,1}), \nu(T_{4,1})), \ldots, unite(\nu(T_{k-1,1}), \nu(T_{k,1}))$$

to produce tree $T_{1,2}, \ldots, T_{k/2,2}$. The designated node $\nu(T_{i,2})$ is chosen to be one of the designated nodes of the subtrees that formed $T_{i,2}$. We call this process of picking the new designated nodes as a subset of the old ones *refining*. The subsequent rounds are done similarly by combining pairs of trees from the previous round and refining designated nodes, until only the tree $T_{1,\lg k}$ remains.

We now make the following observations about this process:

1. All trees $T_{i,r}$ of a given round r have the same number of nodes 2^r.
2. A designated node always has depth at most 2. (This follows from the way *find* does compactions.)
3. A node of depth δ in any of the trees $T_{i,r}$ has at most $\left(\frac{1}{2}\right)^{\delta} \cdot |T_{i,r}|$ successors.

The links in the rounds raise the depth of half the nodes due to (1), and the compactions in the *find* operations of the rounds reduce the average depth of a node in the forest by at most $\frac{1}{4}$ due to (2) and (3). Thus, each round increases the average depth of a node in the forest by at least $\frac{1}{2} - \frac{1}{4} = \frac{1}{4}$. Since there are $\log(k)$ rounds, the proof is complete. □

Combining the previous lemmas yields our best algorithmic lower bound result.

Lemma 7.9.5. *Let $\mathcal{A}$ be a concurrent disjoint set union algorithm obtained by combining linking by DCAS, randomized linking by rank, or linking by random index with find with no compaction, one-try splitting, or two-try splitting. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega\left(m \log\left(\frac{np}{m} + 1\right)\right)$ work. The bound holds in expectation for the linking by random index algorithm even under the independence assumption.*

Proof. We prove the theorem for linking by the DCAS and randomized linking by rank algorithms first. The lower bound is non-trivial only when $m/p < n$. In this case, we describe a particular sequence of operations and schedule that performs the requisite work. Divide the nodes into m/p groups of size $n/(m/p) = np/m$. Lemma 7.9.3 allows us to link each group of nodes into a tree of height $\Omega\left(m \log\left(\frac{np}{m} + 1\right)\right)$. For each such tree, perform *find*(x) on the deepest node x of that tree simultaneously with each of the processes. Now consider the schedule in which processes shadow each other in all the finds. In this schedule, each process does $\Omega\left(m \log\left(\frac{np}{m} + 1\right)\right)$ work per find,

and one find per group. The total number of operations is $m/p \times p = m$, and the total amount of work is $\Omega\left(m \log\left(\frac{np}{m} + 1\right)\right)$.

In the case of the linking by random index algorithm under the independence assumption, we modify the above argument by replacing the use of Lemma 7.9.3 with Lemma 7.9.4, and performing *find*(x) on a uniformly randomly picked node x in the tree. □

Combining the previous lemmas yields our best algorithmic lower bound result.

Theorem 7.9.6. *Let $\mathcal{A}$ be a concurrent disjoint set union algorithm obtained by combining linking by DCAS, randomized linking by rank, or linking by random index with find with no compaction, one-try splitting, or two-try splitting. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega\left(m\left(\log\left(\frac{np}{m} + 1\right) + \alpha\left(n, \frac{m}{np}\right)\right)\right)$ work. The bound holds in expectation for the linking by random index algorithm even under the independence assumption.*

Proof. Combine the results of Lemmas 7.9.2 and 7.9.5. □

As the final algorithmic lower bound, we prove that the independence assumption we have been using to analyze the linking by random index algorithm is indeed necessary. In particular, we present a super-logarithmic work lower bound for the algorithm if the independence assumption does not hold.

Lemma 7.9.7. *Concurrent set union via the linking by random index algorithm performs $\Omega(m\sqrt{p})$ expected work to do $m = n\sqrt{p}$ operations if $\sqrt{p} \leq n$, regardless of which compaction rule the find operations use.*

Proof. We will show an explicit example. Assume $\sqrt{p} < n$, and pick a set S of $\sqrt{p}$ nodes. Let $p/2$ processes attempt to do *unite*(x, y) where each pair of x, y in S is tried by at least one process. The scheduler can wait to see the outcomes of the node comparisons and decide to schedule the processes so that the nodes of S get linked into a linear path of length $\sqrt{p}$. If the remaining $p/2$ processes all perform *find*(x) where x is chosen randomly from S, and are scheduled in lock-step, they will perform, in expectation, $\Omega(\sqrt{p})$ work each, since the expected depth of x is $\sqrt{p}/2$.

Performing the same process on each of the $\lfloor \frac{n}{\sqrt{p}} \rfloor$ sets of nodes leads to $\Omega(np)$ work to do $m = n\sqrt{p}$ operations. The average operation takes $\Omega(\sqrt{p})$ work. □

7.9.2 Problem Lower Bounds

In this subsection, we prove that any concurrent set-union algorithm must do $\Omega(\log \min\{n, p\})$ work for a single operation in the worst case. Furthermore, we build on the worst-case lower bound to show an $\Omega(\log(np/m+1))$ amortized work lower bound for all *symmetric algorithms*, where we say an algorithm is symmetric if:

1. The algorithm's code for the *unite* and *find* procedures does not use process ids.

2. The algorithm does not use the return values of CAS operations.

All our algorithms and all algorithms known to us can be made symmetric without effecting the upper bound analyses of the algorithms. For instance, this can be done if we assume that all CAS operations return false. This does not effect the correctness of our algorithms since we only use the return value of a CAS operation in the *unite* operation to determine if a link has been successful. However, if we do not perform this check atomically, our algorithms remain correct, since a process that has successfully united two trees together will realize this shortly afterwards when its u and v pointers meet at the root of the united tree. The work efficiency analysis increases by at most a factor of two, because we can always imagine the case where processes work in pairs (p, q), and each pair performs operations together and are always scheduled in lock-step. In this case, if p ever performs a successful link CAS$(u.p, u, v)$ and returns, then q's attempt to perform the same link will fail; thus q will only return after it traverses the whole tree and discovers that some other process has already finished the link it wanted to do. The modification we propose to symmetrize the algorithm will simply make p do the same work as q.

Our lower bounds make use of a result on a problem called "wake-up". The *wake-up* problem on k processes asks for a wait-free algorithm with two properties: (i) every process returns a boolean value and at least one process returns true, and (ii) a process may return true only if every process has already executed at least one step. The following lemma is a lower bound on the complexity of wake-up that follows straightfowardly from Jayanti's lower bound in [103].

Lemma 7.9.8 ([102, 103])**.** *For any k process wake-up algorithm that uses variables supporting read, write, and CAS, there is a schedule in which some process performs $\Omega(\log k)$ steps in expectation.*

We solve wake-up via set-union to get our lower bounds below.

Lemma 7.9.9. *The reduction (below) solves the wake-up problem for k processes using a disjoint set union instance with $k+1$ nodes, in which each process executes one unite and two find operations.*

Proof. Let $q_1, \ldots, q_k$ be the k processes and let the nodes be labelled $0, \ldots, k$. The reduction below correctly solves wake-up because:

Reduction q_j's code in wake-up solution.

```
1: procedure WakeUp
2:     unite(j − 1, j); x ← find(0); y ← find(k); return x = y
```

(i) the last process to complete *unite* finds that the leaders of nodes 0 and k are the same and thus returns true, and (ii) no process returns true before all processes have completed *unite*, since no leader of k can be the same as any leader of 0 until they are in the same set, i.e. until the last of the *unite* operations is linearized. □

Theorem 7.9.10. *Let $\mathcal{A}$ be a linearizable wait-free concurrent disjoint set union algorithm using read, write, and CAS. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega(\log \min\{n, p\})$ work in expectation.*

Proof. Instantiate Lemma 7.9.9 with $k = \min\{n-1, p\}$. The most expensive of the three set union operations of the process that performs the most work in the adversarial schedule of Lemma 7.9.8 must do $\Omega(\log \min\{n, p\})$ expected work. □

Corollary 7.9.11. *The disjoint set union algorithm obtained by combining randomized linking with any form of find described in this chapter gives an algorithm with optimal worst-case work per operation up to constant factors when $\log p = \Theta(\log n)$.*

Remark 7.9.12. Theorem 7.9.11 shows that our set union algorithms with randomized linking have optimal work per operation when $p = n^{\varepsilon}$ for constant ε.

Remark 7.9.13. Theorem 7.9.10 establishes a separation in worst-case work complexity between sequential and concurrent set-union when $p = n^{\omega(\frac{1}{\log \log n})}$ since Blum's sequential set-union algorithm has a worst-case work complexity of $O(\frac{\log n}{\log \log n})$ *[25]*.

Lemma 7.9.14. *Let $\mathcal{A}$ be a linearizable wait-free symmetric concurrent disjoint set union algorithm using read, write, and CAS. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}t$ to perform $\Omega(m \log(np/m + 1))$ work.*

Proof. Divide the n nodes into $g = \frac{m}{p}$ groups of size $k+1$, where $k = \frac{np}{8m}$ (disregard any additional nodes); label the groups $G_1, \ldots, G_g$. Note that $m \geq p$ and $m \geq \frac{n}{2}$, so $k \leq \frac{p}{4}$. We divide the $\frac{p}{2}$ (out of the p) processes into two sets $A = \{q_1, \ldots, q_k\}$ and $B = \{q_{k+1}, \ldots, q_{p/2}\}$. Note that $|B| \geq \frac{p}{4}$ and $|A \cup B| = \frac{p}{2}$.

Consider running the wake-up algorithm of Lemma 7.9.9 on the k processes in A using the $k+1$ nodes in G_1. By Lemma 7.9.8 there is a schedule σ_1 in which some process q_i performs $\Omega(\log k)$ steps. Assign to each process in B the same sequence of three set union operations that q_i performs, and define schedule σ_1' to be the schedule σ_1, with the processes of B interleaved in to run in lockstep with q_i. In this schedule, the processes $q_1, \ldots, q_{p/2}$ perform $\Omega(p \log k)$ work to do p set union operations. Repeating this procedure on each group G_j produces schedules σ_j', each of which performs $\Omega(p \log k)$ work. Therefore, in the concatenated schedule of $\sigma_1'\sigma_2' \cdots \sigma_g'$, the processes $q_1, \ldots, q_{p/2}$ perform a total of $\Omega(gp \log(k+1)) = \Omega(m \log(np/m+1))$ work to do a total of $gp = m$ operations. □

Theorem 7.9.15. *Let $\mathcal{A}$ be any linearizable wait-free symmetric concurrent disjoint set union algorithm using read, write, and CAS. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega\left(m\left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n,\frac{m}{np}\right)\right)\right)$ work in expectation.*

Proof. Combine the results of Lemma 7.9.2, whose argument applies to all symmetric algorithms, with Lemma 7.9.14. □

Remark 7.9.16. Theorem 7.9.15 shows that the set union algorithm obtained by combining randomized linking with two-try splitting has optimal amortized work efficiency amongst all symmetric algorithms (up to a constant factor).

The ideas behind our collection of upper and lower bounds lead us to make the following conjecture about the expected work complexity of concurrent disjoint set union.

Conjecture 1. *The expected work complexity of the concurrent set union problem is*

$$\Theta\left(m \cdot \left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n,\frac{m}{np}\right)\right)\right).$$

In light of Theorem 7.8.17, which shows that randomized linking with two-try splitting satisfies the conjectured upper bound, a refutation of Conjecture 1 would imply a more efficient algorithm than randomized linking with two-try splitting. On the other hand, a demonstration of the conjecture would involve proving a universal lower bound; namely, showing that Theorem 7.9.15 holds

for *all* algorithms (as opposed to only symmetric ones). While this chapter proves a universal lower bound on the worst-case complexity of a single operation, it does not prove any universal lower bounds on the total work complexity of m operations. The only such lower bound that is known for the problem is exponentially weaker than the conjectured one. We state this bound, by Jayanti et al., below.

Theorem 7.9.17 ([117]). *Let $\mathcal{A}$ be any linearizable wait-free concurrent disjoint set union algorithm using read, write, and CAS. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega\left(m\left(\log\log\left(\frac{np}{m}+1\right)+\alpha\left(n,\frac{m}{n}\right)\right)\right)$ work in expectation.*

7.10 Remarks and Open Problems

We have presented three linking methods and two splitting methods for concurrent disjoint set union. With any of the linking methods, with or without compaction, the number of steps per operation is $O(\log n)$, worst-case if linking is deterministic, high-probability if randomized. With any of the linking methods and either of the splitting methods, the total work is $O(m(\alpha(n,d)+\log(1+1/d)))$, worst-case if linking is deterministic, average-case if randomized, where the problem density d is $m/(np^2)$ if splitting is one-try, $m/(np)$ if splitting is two-try. No matter what the density, the cost of concurrency is at most a factor of $\log p$, making our algorithms truly scalable. The proofs of the bounds for linking by random index require assuming that the scheduler is non-adversarial, as discussed in Section 7.5.3. The bounds differ for the two splitting methods only for a narrow range of densities: if $m/n = O(1)$ or $m/n = \Omega(p^2)$, the bounds are the same; if $m/n = \omega(1)$ and $m/n = o(p^2)$, the bounds differ by a factor of at most $\log p$.

The $O(\log n)$ step bound is tight for all our algorithms. The work bounds for splitting are almost tight: any symmetric algorithm (as defined in Section 7.9) has a work bound of $\Omega\left(m\cdot\left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n,\frac{m}{np}\right)\right)\right)$. We conjecture that the same lower bound can be shown for asymmetric algorithms also (Conjecture 1), but leave the proof or refutation of this statement as an open problem.

Our results leave open the question of whether there is an efficient *deterministic* algorithm that uses only CAS: our deterministic algorithm uses DCAS. Recently we have developed a surprisingly simple algorithm that answers this question positively. The algorithm combines two ideas: the use of *latent links*, which represent unites started but not finished, and *deterministic coin tossing* [36], which provides a deterministic way to break ties. The worst-case and amortized time bounds for

finds are the same as those in the present work; the bounds for unites are larger by a factor of $\lg^* n$, reducible to $\lg^* p$. We shall describe this result in a forthcoming paper.

In some applications of disjoint set union, such as computing flow graph information [193, 194] each set has a name or some other associated value, such as the number of elements in the set. We can extend the compressed tree date structure to support set values by storing these in the set roots. In the sequential setting, it is easy to update set value information in $O(1)$ time during a link. But in the concurrent setting, updating the value in the new root during a link requires a DCAS or some more-complicated implementation using CAS. Updating root values using DCAS invalidates our analysis. Consider n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. Suppose $p = n$, and unite(1, n), unite(2, n),..., unite($n - 1$, n) are performed concurrently using linking by rank via DCAS. Assume the tie-breaking total order is numeric. At most one link will succeed initially, say the link of 1 and n. After this link, all nodes except n will still have rank 0, and n will have rank 1. The algorithm of Section 7.5.1 does all the remaining links concurrently using CAS, since none affects node n. But if each such link needs to update the value in node n, the remaining links must be done one-at-a-time, resulting in overall work $\Omega(np)$.

We think this problem can be overcome, and that the concurrent set union problem with set values can be solved in a work bound that is quasi-linear in m and poly-logarithmic in p. But doing so may well require relaxing the linearization requirement: instead of continuing to try to link each node i and n, suppose the algorithm does a different set of links that reduce the contention. For example, the algorithm could link roots in pairs, then the remaining roots in pairs, and so on. The set resulting from all the links would be the same, but the intermediate sets would not correspond to any linearization of the original unites. Even though it violates linearization, such an algorithm might suffice in many if not all applications.

An algorithm of this kind needs a mechanism to restructure the links. We think some sort of binary tree structure, like the one used by Ellen and Woelfel [57] in their fetch-and-increment algorithm but more dynamic, may suffice. We leave open the development of this idea or some other idea to solve the problem of concurrent sets with values.

Although our results are for a shared memory model, we think they will fruitfully extend to a distributed-memory, message-passing setting.

Our work is theoretical, but others [3, 51, 95] have implemented some of our algorithms on CPUs and GPUs and experimented with them. On many realistic data sets, our algorithms run as fast or faster than all others.

Chapter 8

Fast Arrays and their Applications

8.1 Introduction

Arrays are the most fundamental data structure in computer science. Semantically, an *array* of length m is an object that supports the following interface:

- INITIALIZE(m, f): return an array $\mathcal{O}$ initialized to $\mathcal{O}[i] = f(i)$ for each $i \in [m]$.[1]
- $\mathcal{O}$.READ(i): return $\mathcal{O}[i]$, if $i \in [m]$.
- $\mathcal{O}$.WRITE(i, v): update $\mathcal{O}[i]$'s value to v, if $i \in [m]$.

Here, INITIALIZE() is the *constructor* method that creates the object, and READ() and WRITE() are the regular operations an array supports. Ordinarily, initialization is achieved by allocating an array of length m and looping through to initialize its entries, while reads and writes simply use the hardware load and store instructions. This standard implementation achieves a space complexity of $O(m)$, and time complexities of $O(m)$ for initialization and $O(1)$ for reads and writes. These time complexities are good for applications that eventually access most of the entries of the array. But, some applications—such as adjacency matrix representations of sparse graphs, van Emde Boas trees, and certain hash tables—need to allocate a large array when only a small fraction of the array will eventually be accessed. The time complexities of such algorithms would improve drastically if we had *fast arrays*: arrays that support all three operations—READ(), WRITE(), and *even* INITIALIZE()—in just $O(1)$ worst-case time. Perhaps surprisingly, sequential fast array implementations have been known for decades, but, to the best of our knowledge, concurrent implementations do not exist. We design the first algorithms for concurrent fast arrays in this

[1] For a positive integer m, we use the notation $[m] \triangleq \{0, 1, \ldots, m-1\}$.

chapter.

8.1.1 Sequential fast arrays: history and applications

Sequential algorithms for fast arrays date back to at least the 1970s [2, 151, 19]. In fact, the well known *folklore algorithm* for the problem (which we will revisit in Section 8.3) was alluded to in an exercise of the celebrated text by Aho, Hopcroft, and Ullman [2] and further described by Mehlhorn [151] and Bentley [19]; it achieves a deterministic worst-case time complexity of $O(1)$ for each of the three operations, while using only $3m + 1$ memory words. Fast arrays have been important to the efficiency of several algorithms. Notably:

- Fast arrays are used in implementations of *van Emde Boas trees* [40, 161]—associative arrays that store keys from a universe $\{1, 2, \ldots, u\}$ and support *insert*, *get*, and *delete* with a time complexity of just $O(\log \log u)$.

- Katoh et al. [127] note that Knuth employs fast arrays in the implementation of the hash table in his *Simpath* algorithm [131], which enumerates all simple paths between two vertices in a graph. Knuth uses the hash table to efficiently implement a certain data structure called ZDD (Zero-suppressed binary Decision Diagram) [157], which has many applications besides Simpath [191, 156, 100, 155, 212, 131].

- When a sparse graph of n vertices and $m \ll n^2$ edges is represented using an adjacency matrix, mere initialization can take $\Theta(n^2)$ time with a traditional array. With a fast array however, the graph can be stored in just $O(m)$ time. Consequently, for a constant degree graph, storing the graph takes $O(n)$ time instead of $\Theta(n^2)$ time.

More generally, fast arrays can increase the asymptotic efficiency of algorithms that have higher space complexity than time complexity—just allocate all the space in one huge block in $O(1)$ time.

This range of applications has spurred a lot of research into fast arrays in recent years. A string of papers, starting with Navarro's work in 2012 and culminating in three back to back papers in 2017 by Hagerup and Kammer, Loong et al., and Katoh and Goto, have brought down the space complexity from $3m + 1$ to $m + 1$ using complex bit-packing and chaining techniques [163, 164, 85, 146, 127]. Fredriksson and Kilpeläinen recently studied the empirical running times of the more practical implementations of these sequential fast arrays [65].

8.1.2 Concurrent fast arrays

In contrast to sequential fast arrays, which have been well studied, there has been no prior work on concurrent fast arrays, to the best of our knowledge. In this chapter, we propose and design algorithms for two variants of concurrent fast arrays:

- Fast Array: This is an implementation of an array which supports the standard operations—INITIALIZE(m, f), READ(i), and WRITE(i, v)—and satisfies two conditions. First, each operation is *linearizable*, i.e., it appears to take effect at some instant between its invocation and response [93]. Second, each operation is not only *wait-free* [89], but the process that executes the operation completes it in a constant number of its steps. The first condition ensures *atomicity*, and the second condition ensures *efficiency*.

- Fast Generalized Array: Besides *load* and *store*, modern architectures like x86 commonly support read-modify-write (RMW) primitives, such as Compare-and-Swap (CAS), Fetch-and-Add (FAA), and Fetch-and-Store (FAS) [98]. In fact, some of these primitives are indispensible for efficiency and even solvability of problems that arise in concurrent systems. For instance, implementing a wait-free queue is impossible using only loads and stores [89]. Mutex locks can be implemented using loads and stores, but constant RMR (remote memory reference) complexity implementations are impossible using only loads and stores [17, 42, 152].

  Since RMW primitives are supported by hardware and are essential for concurrent algorithms, it would be ideal if the components of the fast array can be manipulated using these primitives. For instance, when implementing a fast array $\mathcal{O}$ on a multiprocessor that supports CAS and FAS in hardware, a process π should not only be able to read $\mathcal{O}[i]$ and write to $\mathcal{O}[i]$, but should also be able to CAS $\mathcal{O}[i]$ and FAS $\mathcal{O}[i]$. We term such an array, which allows all hardware-supported operations to be applied to its components, a *generalized array*.

  Let $\mathcal{S}$ be the set of hardware-supported RMW primitives. A *fast generalized array* is an implementation that not only supports $O(1)$-time linearizable INITIALIZE(m, f), READ(i), and WRITE(i, v) operations, but also supports $O(1)$-time linearizable operations from the set $\mathcal{S}$.

8.1.3 Our contributions

In addition to defining the two types of concurrent fast arrays, our work makes the following two principal contributions:

- We design an algorithm for the (standard) fast array. If p processes share a fast array of length m, our algorithm uses only $O(m+p)$ space. More generally, to instantiate and use k fast arrays (for any k) of lengths $m_1, \ldots, m_k$, our algorithm uses only $O(M+p)$ space, where $M = \sum_{j=1}^{k} m_j$.
- We enhance the above algorithm to design a fast generalized array. Its space complexity is the same as the previous algorithm's—$O(m+p)$ for a single array of length m, and $O(M+p)$ for multiple arrays of combined length M.

Both of the above algorithms require hardware support for read, write, and CAS.

8.2 Model

We work in the standard asynchronous shared memory multiprocessor model where p processes, numbered $0, \ldots, p-1$ run concurrently but asynchronously, and each process is either performing an initializable array operation or is idle. The computation proceeds in steps, where an *adversarial scheduler* decides which process π takes the next step.

To provide synchronization, we assume the hardware compare-and-swap (CAS) synchronization primitive. The CAS operation on a memory word X with arguments *old* and *new* is called as follows: CAS(X, old, new). The operation is atomic and has the following behavior. If $X = old$, then X's value is updated to *new* and *true* is returned to indicate that the operation successfully changed the value; otherwise, if $X \neq old$, the value of X is not changed and *false* is returned. On modern x86 architectures, individual memory words are 64-bits, and so any hardware primitive can be applied on a standard 64-bit word. Usefully however, CAS can also be executed on 128-bit double-words, i.e., two adjacent words of memory [98]. We will make use of this feature. (Note that this 128-bit CAS operation is *not* DCAS—a primitive that does CAS on two non-adjacent memory locations, which is not supported in modern architectures.)

A data structure is *linearizable* if each operation can be assigned a unique *linearization point* between its invocation and return, and the return values of the operations are consistent with those of a sequential execution in which operations are executed in the order of linearization points [93].

Operations are *bounded wait-free* if there exists a bound b such that every invocation by a process π returns within b of π's own steps [89]. In the literature, data structures that are both linearizable and wait-free are called *atomic*.

We measure the efficiency of an algorithm by its worst-case work and space complexities. The *space complexity* of an algorithm is the total number of memory words that the algorithm uses. The *work complexity* of an operation by a process π, is the total number of steps executed by π between the invocation and return of that operation. Since work complexity is the natural generalization of time complexity to multiprocessors, it is often called *time complexity* in the literature; we adopt this convention and use the two terms interchangeably. Furthermore, as is standard [2, 151, 19, 163, 164, 161, 182], we assume that it takes constant time to allocate an *uninitialized* array of any size n. We call an object implementation *fast* if every operation on that object takes only $O(1)$ time to execute in the worst-case. Our work focuses on fast algorithms for arrays and generalized arrays.

8.3 Folklore Sequential Algorithm

Our concurrent algorithms are inspired by the folklore sequential algorithm, and so we present the pseudo-code for a folkore fast array object $\mathcal{O}$ in Algorithm 23, and describe it below.

Algorithm 23 The folklore algorithm for a sequential fast array.

```
    procedure INITIALIZE(m, f)
1:      A ← new array[m]
2:      B ← new array[m]
3:      C ← new array[m]
4:      f_init ← f
5:      X ← 0
    procedure READ(i)
6:      if 0 ≤ B[i] < X and C[B[i]] = i then return A[i] else return f_init(i)
    procedure WRITE(i, v)
7:      A[i] ← v
8:      if B[i] < 0 or B[i] ≥ X or C[B[i]] ≠ i then
9:          C[X] ← i
10:         B[i] ← X
11:         X ← X + 1
```

The method INITIALIZE(m, f) instantiates a new fast array $\mathcal{O}$ of length m. The implementation of the fast array uses three un-initialized arrays A, B, and C, each of length m, an integer X, and

stores a pointer f_{init} to the function f. We call A the *principal array* and use $A[i]$ to hold the current value of the abstract element $\mathcal{O}[i]$ for each index i that has been *initialized*, i.e., written to at least once. The elements of $A[i]$ corresponding to uninitialized indices of $\mathcal{O}[i]$ hold their initial, arbitrary values. B, C, and X are used to keep track of which indices i have already been initialized (as we describe later).

Using the mechanism described above, implementing read and write becomes simple. READ(i) just returns $A[i]$ if i has been initialized, and $f(i)$ otherwise. Correspondingly, WRITE(i, v) simply writes $A[i] \leftarrow v$, and ensures that index i is marked as initialized.

The main difficulty of the algorithm lies in efficiently remembering which set of indices i have been initialized. We use the array C and the integer X to maintain this set as follows. If k indices have already been initialized, then we ensure that $X = k$ and that the sub-array $C[0, \ldots, X-1]$ holds the values of these initialized indices. Correspondingly, we spend constant time in the INITIALIZE() method to set $X \leftarrow 0$. Terminologically, we call C the *certification array*, call the elements of the array *certificates*, call the elements of the sub-array $C[0, \ldots, X-1]$ *valid*, and say that an index i is *certified* when it appears in the valid sub-array.

Maintaining A, C, and X is sufficient to get a *correct* implementation of an array, but not an *efficient* one. For efficiency, we need to distinguish between certified and un-certified indices in constant time. We use the array B for this purpose. In particular, whenever we certify a new index i in an element $C[j]$, we set $B[i] \leftarrow j$ to maintain the invariant that

$$\mathcal{I} \equiv \forall i \in [m],\ (0 \leq B[i] < X \text{ and } C[B[i]] = i \textit{ if and only if } \text{index } i \text{ is initialized}).$$

The check that $0 \leq B[i] < X$ ensures that $C[B[i]]$ is valid, while the check that $C[B[i]] = i$ ensures that this valid element of the certification array, indeed, certifies that index i is initialized.

8.4 Our Concurrent Fast-Array

The goal of this section is to design a linearizable wait-free fast-array that is both time and space efficient. We do so by building on the ideas of the folklore algorithm.

The folklore algorithm is built on two pillars: (1) the principal array A, which stores the values of initialized indices, and (2) the *certification mechanism* constituted by B, C, and X, which ensures that initialized indices can be identified in constant time by invariant $\mathcal{I}$. The principal array can easily be maintained in the concurrent setting, however the certification mechanism, which is the main workhorse of Algorithm 23, must be redesigned to cope with concurrency.

The difficulty of using the sequential certification mechanism with multiple processors stems from the contention on the variable X, and on the next available slot in the certification array, $C[X]$. In particular, if all p processors are concurrently performing different write operations on different un-initialized indices $i_0, \ldots, i_{p-1}$, then the old certification mechanism will direct all of them to the same location $C[X]$ in the certification array. Regardless of how the contention is resolved, only one index can fit into $C[X]$, meaning that $p-1$ processes will fail to certify their index by placing it in $C[X]$ and will thereby need to find an alternate location in the certification array. So, in the worst-case, only one process will finish its operation after all p processes do one unit of work each, meaning the algorithm will do $O(p)$ work per operation rather than $O(1)$.

We overcome this difficulty of adapting the certifying mechanism posed by contention on C and X by introducing four ideas that we detail below. Our first idea will eliminate the contention, thereby enabling constant time certifications and look-ups; however, it bloats the space complexity to $\Omega(m \cdot p)$. Our second and third ideas, in combination, eliminate this space overhead and bring the space complexity down to just $O(m+p)$, while ensuring that the time complexities of operations remain at just $O(1)$. Our fourth idea describes how to share resources in order to minimize the space complexity when multiple fast arrays are instantiated.

Individual certification arrays: our first idea is to eliminate the universal C and X, and instead equip each process π with its own certification array c_π and a corresponding *control variable* $\mathrm{X}[\pi]$. Here, X is a one-dimensional array of length p that is indexed by process ids, and each c_π is an array of length m. Thus, process π certifies a new index i by performing three steps: (1) writing $c_\pi[\mathrm{X}[\pi]] \leftarrow i$, (2) setting $\mathrm{X}[\pi] \leftarrow \mathrm{X}[\pi] + 1$, and (3) writing $B[i] \leftarrow (\pi, \mathrm{X}[\pi] - 1)$. (While it will not yet be clear to the reader at this stage, the relative order of steps 2 and 3 is very important for the correctness of later ideas. We expand on this thought in the forthcoming Remark 8.4.1.) Unlike the act of certifying a new index which involves modifying certification arrays and control variables, the act of checking whether a given index i is certified only requires reading. We allow process π to freely read the arrays of other processes while checking if an index is certified. This idea of individual certification arrays by itself would lead to a concurrent fast-array algorithm; however, the space complexity of this algorithm is inherently super-linear. In particular, if all p processes concurrently write to a previously un-initialized location i, an adversarial scheduler can force each of them to certify that location in its own certification array; if this happens for each of the m indices, then each c_π must store m indices, leading to a total space complexity of $\Theta(mp)$.

Synchronization and walk-back: to reduce the space complexity induced by individual

certification arrays, we must ensure that each index i is certified by at most one process, even if multiple processes perform concurrent writes to the same un-initialized index. To do this, we introduce two related ideas: *synchronization on B* and *walk-back*. That is, each processor π that wishes to certify an index i attempts to CAS (rather than write) the pair $(\pi, \mathrm{X}[\pi] - 1)$—indicating the location in its certification array where i is certified—into $B[i]$. We orchestrate the update to $B[i]$ using CAS to ensure that at most one process gets a return value of *true*, indicating that *it* is the process that succeeded in certifying i. Each other process π, whose CAS to $B[i]$ fails, "walks back", i.e., it reclaims the location $c_\pi[\mathrm{X}[\pi] - 1]$ that it was going to use to certify i by decrementing $\mathrm{X}[\pi]$. Since each index is certified by at most one process, and each process has at most one certification location that it will walk back on at any given time, the total space used across all c_π arrays is $O(m + p)$.

Array doubling: our synchronization and walk-back scheme guarantees that at most $O(m+p)$ space is *used* across all of the c_π arrays, but we do not *a priori* know how many locations each process π will use in its c_π array. To ensure that we allocate only as much space as we use, we employ the classic idea of *array-doubling* from sequential algorithms. We initially allocate constant sized c_π arrays. Each time c_π fills up, we replace it with a newly allocated array c'_π of length $2 \cdot c_\pi.len$ and copy over the old $c_\pi.len$ elements from c_π to c'_π. Note that we *do not* de-allocate the old array c_π when we switch to c'_π, since other processes could be accessing it; yet, since the sum of a geometric series is proportional to the largest term in the series, our total memory allocation for the c_π arrays is proportional to the amount of space we end up using. (Note that it is important to have a mechanism by which other processes can get access to the current array c_π, since the location of the array is changing whenever we double. We describe this detail when we discuss the pseudo-code in a later sub-section. We will also describe how to implement array doubling with worst-case, rather than amortized, constant time per operation in the same sub-section.)

Sharing the certification mechanism: Array doubling allows us to share a single certification mechanism across all fast-array objects that we initialize. In particular, if we have multiple fast-arrays $\mathcal{O}_1, \ldots, \mathcal{O}_k$, each $\mathcal{O}_j$ can simply maintain the two instance variables $\mathcal{O}_j.A$ and $\mathcal{O}_j.B$, and *share* the certification mechanism—$\forall \pi \in [p], (\mathrm{X}[\pi], c_\pi)$. All we need to do to enable this sharing is store a pointer $\&(\mathcal{O}_j.A[i])$ as the certificate that i is initialized in fast-array $\mathcal{O}_j$, rather than just store the index i. Since all fast array objects can share one certification mechanism, the space complexity of maintaining k fast-arrays $\mathcal{O}_1, \ldots, \mathcal{O}_k$ of sizes $m_1, \ldots, m_k$ is just $O(M + p)$, where $M \triangleq \sum_{j=1}^{k} m_j$.

Remark 8.4.1 (the relative order of synchronizing and incrementing)**.** When a process π, with next available certification location $x_\pi = \mathrm{X}[\pi]$, is certifying a new index i, we described that our algorithm follows three logical steps (not including potential walk-back): (1) *writing the certificate*: $c_\pi[x_\pi] \leftarrow i$, (2) *incrementing* $\mathrm{X}[\pi]$: $\mathrm{X}[\pi] \leftarrow x_\pi + 1$, and (3) *synchronizing* on $B[i]$: attempting to CAS the value (π, x_π) into $B[i]$. At first glance, it may appear that step (3) can be executed before step (2). Indeed, if this were possible, then we could simplify our algorithm by avoiding walk-backs altogether, since a process π that fails the CAS could simply not increment $\mathrm{X}[\pi]$. However, as we mentioned earlier, the relative order of steps (2) and (3) is pivotal to correctness. We now explain why.

Consider a scenario where two processes π and τ are each performing the operation $\textsc{Write}(i, new)$ on a previously un-initialized location $\mathcal{O}[i]$ whose initial value is $\mathcal{O}[i] = f(i) = old$, with the order of steps (2) and (3) swapped. Then the following sequence of events can occur:

1. Both process π and τ write $A[i] \leftarrow new$, read the same old value $b \leftarrow B[i]$ (in anticipation of having to $\textsc{Cas}$ $B[i]$ in step (3)), and start the certification process.

2. π completes steps (1) and (3) and thereby successfully changes $B[i]$'s value to (π, x_π). However, i is still not certified since step (2) is yet to be done, and $\mathrm{X}[\pi] \not> x_\pi$.

3. τ completes steps (1) and (3), but because τ's $\textsc{Cas}$ in step (3) fails, it does not need to execute step (2), and it returns from its write operation.

4. Having finished its write operation, τ performs $\textsc{Read}(i)$, but sees that i is not yet certified (since π has not yet finished step (2)), and returns $f(i) = old$.

- This execution is not linearizable, since τ reads the value old in $\mathcal{O}[i]$ even after it finishes writing new.

Remark 8.4.1 establishes that a process π must increment $\mathrm{X}[\pi]$ before it synchronizes at $B[i]$ in the certification process. This means that we must indeed implement walk-back to achieve space efficiency. However, if walk-back is not implemented very carefully, it can lead to a nasty race condition. We describe this possible race condition, and how to overcome it, in the next subsection.

8.4.1 A tricky race condition that must be avoided

Until now, we have described the main ideas that propel our space-efficient fast-array implementation at a high-level. Of these ideas, individual certification arrays are straightforward to implement as suggested, and array-doubling requires only mild adaptation to work in the face of asynchronous concurrency. The idea of walk-back however can lead to a nasty race-condition if it is not implemented correctly. We describe this potential race, and how we overcome it below.

In order to understand the race condition, let us consider the following set-up. We have a freshly initialized fast-array $\mathcal{O}$ with just two locations $\mathcal{O}[0,1]$, and initialization function $f(i) = old$. There are three processors π, τ, and ρ with: $\mathrm{X}[\pi] = 0$, $\mathrm{X}[\tau] = 0$, and $\mathrm{X}[\rho] = 0$. The processes will perform the following operations:

- π will perform $\mathcal{O}.\textsc{Write}(0, new)$ followed by $\mathcal{O}.\textsc{Write}(1, new)$
- τ will perform $\mathcal{O}.\textsc{Write}(0, new)$
- ρ will perform $\mathcal{O}.\textsc{Read}(0)$ followed by another $\mathcal{O}.\textsc{Read}(0)$

By design, both locations initially hold the value *old* and at some point in time will take on the value *new* and hold that value forever. However, the race condition will be that ρ's first read of index 0 will return *new*, while its second read will return *old*. The initial value of $B[0]$—which can be arbitrary by design of the algorithm—is pivotal to achieving the race. In particular, we consider the initial value $B[0] = (\pi, 0)$. The initial values of $A[0,1]$ and $B[1]$ can be arbitrary.

We describe the offending run below in a sequence of bullet points. When the relative order of certain operations do not matter, we may describe them all in the same bullet point.

- Recall that $B[0] = (\pi, 0)$, π is performing $\textsc{Write}(0, new)$, and τ is performing $\textsc{Write}(0, new)$.

1. π and τ both write $A[0] \leftarrow new$, and both read the initial value b of $B[0]$. (They will need this value b when they attempt to certify index 0 and do a $\textsc{Cas}$ on $B[0]$ later.)
2. π and τ both conclude that index 0 is not certified yet, and thus wish to certify the location.

- Recall that $\mathrm{X}[\pi] = 0$.

3. π's next open certification location is 0, thus π writes $c_\pi[0] \leftarrow 0$, and increments $\mathrm{X}[\pi] \leftarrow 1$. π now stalls (before attempting to $\textsc{Cas}$ its certification location $(\pi, 0)$ into $B[0]$).

- Notice that while π is not finished with its certification process, index 0 is already certified, since $B[0] = (\pi, 0)$ initially, and location $c_\pi[0]$ is valid and holds the value 0.

4. ρ does its first READ(0) operation. That is, it reads $B[0] = (\pi, 0)$, checks that $c_\pi[0]$ is valid and that $c_\pi[0] = 0$ and thereby returns the value $A[0] = new$.

5. ρ starts its second READ(0) operation. It starts its verification by reading $B[0] = (\pi, 0)$, and then stalls.

6. τ's next open certification location is 0, thus τ writes $c_\tau[0] \leftarrow 0$, increments $\mathrm{X}[\tau] \leftarrow 1$, and successfully CASes its certification location $(\tau, 0)$ into $B[0]$.

- Notice that index 0 is now certified by two certificates $c_\pi[0]$ and $c_\tau[0]$. $B[0] = (\tau, 0)$ identifies only the new certificate, but process ρ is about to check for the old certificate $c_\pi[0]$.

7. π attempts to finish its certification process by CASing $(\pi, 0)$ into $B[0]$. However, its CAS fails. Thus, π walks-back, and resets $\mathrm{X}[\pi] \leftarrow 0$. This completes π's write operation to index 0.

8. π does its entire WRITE$(1, new)$ operation. That is, it writes $A[1] \leftarrow new$, writes $c_\pi[0] \leftarrow 1$, increments $\mathrm{X}[\pi]$ to 1, successfully CASes $(\pi, 0)$ into $B[1]$, and returns.

9. ρ now finishes its operation. Since it had previously read $B[0] = (\pi, 0)$ and $\mathrm{X}[\pi] = 1 > 0$, it checks $c_\pi[0]$, finds the value 1 there, concludes that index 0 is not certified, and thereby returns $f(0) = old$.

- This run cannot be linearized since $\mathcal{O}[0]$ was initially *old* and became *new*, but ρ reads its value to be *new* and subsequently re-reads the value to be *old*.

We observe that the cause of this race condition is the coincidental initial value of $B[0]$. In particular, $B[0]$'s (potentially arbitrary) initial value, happened to coincide with the exact location that process π would use to certify index 0 and later have to walk-back on. This coincidence, in turn, caused $B[0]$ to become certified during step (3), before π finished its certification operation by updating $B[0]$ with a CAS.

Tombstoning: since we have only constant time to initialize $\mathcal{O}$, we *cannot* control the initial values of all the elements $B[i]$. However, we *can* control which location in the certification array π uses to certify $B[i]$. Therefore, we eliminate this nasty race condition as follows. If the initial value of $B[i]$ is (π, k), then we ensure that that particular process π does not attempt to use its certificate $c_\pi[k]$ to certify index i. If it so happens that $c_\pi[k]$ is the next available certificate for process π when process π is attempting to certify i, then we simply *tombstone* that location by writing a special null-value $c_\pi[k] \leftarrow \perp$, and use the next location $c_\pi[k+1]$ to certify i. This ensures correctness.

Furthermore, observe that for each location i, there is exactly one initial value $B[i] = (\pi, k)$ that references exactly one process π and one specific location k. Thus, at most m locations get tombstoned by our method across all processes, and the space complexity bound of $O(M + p)$ continues to hold true even in the worst-case. (We expect that tombstoning will occur only very rarely in practice.)

8.4.2 The pseudo-code and its description

Having described individual certification arrays, synchronization and walk-back, array-doubling, and tombstoning, we are ready to describe our fast-array algorithm. We present the pseudo-code as Algorithm 24, and describe it below.

Naming conventions: In order to distinguish between variables of different processes and operations that are performed by a particular process π, we use subscripts. For example, we denote a local variable x of process π by x_π, and denote a READ() operation by process π as $\text{READ}_\pi()$. We use captial letters, such as A and X, to refer to arrays that all processes have the address of by default. Importantly, note that the pointer to the *current* certification arrays, c_π, follows the above convention, and by default only process π has access to the array. In order to allow other processes to access these arrays, our implementation stores a pair in the control variable $\text{X}[\pi]$. So, initially $\text{X}[\pi] = (0, c_\pi)$ (rather than $\text{X}[\pi] = 0$).

In order to implement array doubling, we maintain a *next* certification array c'_π alongside the current array c_π. As such, initially $\text{X}[\pi] = (0, c_\pi)$ and no process other than π has access to the array pointed to by c'_π. When c_π fills up entirely, we maintain the invariant that $c'_\pi[0, \ldots, c_\pi.len - 1] = c_\pi[0, \ldots, c_\pi.len - 1]$ and thus, we can simply replace $\text{X}[\pi] = (x_\pi, c_\pi)$ by $\text{X}[\pi] = (x_\pi, c'_\pi)$; we also rename c'_π as c_π because it has become the current array, and allocate a new (un-initialized) c'_π that is twice the length of the new current array. In order to ensure that $c'_\pi[0, \ldots, c_\pi.len - 1] = c_\pi[0, \ldots, c_\pi.len - 1]$ by the time c_π gets entirely full, we *transfer* two values from c_π to c'_π each time a new value is appended to c_π. We note that we *do not* de-allocate the old certification arrays because other processes could potentially be reading from them—this does not change the asymptotic space complexity. However, each process can store pointers to all of its arrays so that they can be de-allocated when the fast array is no longer needed. In our algorithm, we choose to start with a c_π of length 2. (Any other constant length would have sufficed.)

A line-by-line description of the code is as follows. $\mathcal{O}.\text{INITIALIZE}_\pi(m_\pi, f_\pi)$ simply allocates the unintialized arrays A and B of length m_π (Lines 1 and 2), and stores the initialization function f_π

Algorithm 24 Atomic fast array for p processes. Pseudo-code shown for an arbitrary process π.

Variables:

For each process $\pi \in [p]$ the following variables are shared across *all* fast-arrays $\mathcal{O}$:

- $c_\pi[0,1]$ is a pointer to an allocated un-initialized array of length 2.
- $c'_\pi[0,\ldots,3]$ is a pointer to an allocated un-initialized array of length 4.
- k_π is a non-negative integer that is initialized to 0.
- $\mathrm{X}[\pi]$ is a pair that is initialized to $(0, c_\pi)$.

Each object $\mathcal{O}$ has three instance variables instantiated by $\textsc{Initialize}_\pi(m_\pi, f_\pi)$:

- A and B are arrays of length m_π.
- f_{init} stores the initial value function.

Each process $\pi \in [p]$ uses the following arbitrarily initialized temporary local variables:

- b_π, b_π^{old}: hold (process id, array index) pairs.
- x_π: holds an array index.
- c_π^{other}: holds an array pointer.

procedure $\mathcal{O}.\textsc{Initialize}_\pi(m_\pi, f_\pi)$
1: $A \leftarrow$ **new** array$[m_\pi]$
2: $B \leftarrow$ **new** array$[m_\pi]$
3: $f_{init} \leftarrow f_\pi$

procedure $\mathcal{O}.\textsc{Write}_\pi(i_\pi, v_\pi)$
4: $A[i_\pi] \leftarrow v_\pi$
5: $\textsc{Certify}_\pi(i_\pi)$

procedure $\mathcal{O}.\textsc{Read}_\pi(i_\pi)$
6: **if** $\textsc{IsCertified}_\pi(i_\pi)$ **then**
7: **return** $A[i_\pi]$
8: **else return** $f_{init}(i_\pi)$

9: **procedure** $\mathcal{O}.\textsc{IsCertified}_\pi(i_\pi)$
10: $b_\pi \leftarrow B[i_\pi]$
11: **if** $0 \le b_\pi.pid < p$ **then**
12: $(x_\pi, c_\pi^{other}) \leftarrow \mathrm{X}[b_\pi.pid]$
13: **if** $0 \le b_\pi.loc < x_\pi$ **and** $c_\pi^{other}[b_\pi.loc] = \&A[i_\pi]$ **then return** *true*
14: **return** *false*

15: **procedure** $\mathcal{O}.\textsc{Certify}_\pi(i_\pi)$
16: $b_\pi^{old} \leftarrow B[i_\pi]$
17: **if** $\textsc{IsCertified}_\pi(i_\pi)$ **then return**
18: $(x_\pi, -) \leftarrow \mathrm{X}[\pi]$
19: **if** $x_\pi \ge c_\pi.len$ **then**
20: $c_\pi \leftarrow c'_\pi$
21: $c'_\pi \leftarrow$ **new** array$[2 \cdot c_\pi.len]$
22: $k_\pi \leftarrow 0$
23: **if** $b_\pi^{old}.pid = \pi$ **and** $b_\pi^{old}.loc = x_\pi$ **then**
24: $c_\pi[x_\pi] = \perp$
25: $x_\pi \leftarrow x_\pi + 1$
26: $c_\pi[x_\pi] \leftarrow \&A[i_\pi]$
27: $\mathrm{X}[\pi] \leftarrow (x_\pi + 1, c_\pi)$
28: **while** $k_\pi < 2x_\pi - c_\pi.len$ **do**
29: $c'_\pi[k_\pi] \leftarrow c_\pi[k_\pi]$
30: $k_\pi \leftarrow k_\pi + 1$
31: **if not** $\textsc{Cas}(B[i_\pi], b_\pi^{old}, (\pi, x_\pi))$ **then**
32: $\mathrm{X}[\pi] \leftarrow (x_\pi, c_\pi)$

as f_{init} for future use (Line 3). The elements of $A[i]$ will be used to hold the values of the abstract elements $\mathcal{O}[i]$. The elements of $B[i]$ will be used to hold process-index pairs (π, j); as a matter of convention, we call the first element in the pair $B[i].pid$ (process id) and the second element $B[i].loc$ (location).

$\textsc{Write}_\pi(i_\pi, v_\pi)$ first updates $A[i_\pi]$ to the new value v_π (Line 4). Since index i_π may not yet be certified, it calls $\textsc{Certify}_\pi(i_\pi)$ (Line 5). As we describe below, $\textsc{Certify}_\pi(i_\pi)$ only creates a new certificate for i_π if the index is not already certified.

Since $\textsc{Certify}_\pi(i_\pi)$ creates a new certificate (if necessary), and must perform the synchronization-CAS on $B[i_\pi]$ after creating a certificate, it must read the old value of $B[i_\pi]$. Thus, $\textsc{Certify}_\pi(i_\pi)$ starts by reading $b_\pi^{old} \leftarrow B[i]$ (Line 16). Certification should only be done if i_π is not already certified, and so it calls $\textsc{IsCertified}_\pi(i_\pi)$ and returns immediately if location i_π is already certified (Line 17). Otherwise, it fetches the next location x_π that is free for creating a certificate (Line 18). Certification proceeds in five steps:

1. If the current certification array is full (check on Line 19), then the next array becomes the current one (Line 20), and a new (un-initialized) next array is allocated (Line 21). Finally, the local variable k_π—which is used to keep track of how many elements in the current array have already been transferred to the next array—is reset to 0 (Line 22).

2. At this point, we are sure that location $c_\pi[x_\pi]$ exists, and is free to certify a new index. Lines 23–25 tombstone this location and increment x_π if $B[i]$'s initial value happens to already hold the value (π, x_π). (This step eliminates the nasty race condition described earlier.)

3. Lines 26–27 certify location $A[i_\pi]$, by writing a pointer to $\&A[i_\pi]$ in $c_\pi[x_\pi]$ and updating the current array pointer and the length of the valid sub-array values in $\mathrm{X}[\pi]$.

4. Since a new location has been filled up in c_π, we must transfer values from c_π to c'_π. If there were no walking-back (described in the next step), then we would need to transfer exactly two values. However, because of potential walk-backs (in previous operations), it is possible that no values need to be transferred in this operation. Lines 28–30 orchestrate this transfer by maintaining k_π's progress relative to x_π.

5. Finally, π performs the synchronization step: it tries to finish its certification process with a Cas on Line 31. If the Cas fails, then it walks-back on Line 32.

That completes the description of $\textsc{Certify}_\pi(i_\pi)$.

$\textsc{Read}_\pi(i_\pi)$ simply checks whether element i_π is certified (Line 6), and returns $A[i_\pi]$ or $f_{init}(i_\pi)$

accordingly (Lines 7–8).

Like the read operation, IsCertified$_\pi(i_\pi)$ is also short. However, its code highlights an important point. The operation starts by reading $b_\pi \leftarrow B[i_\pi]$ which would hold the location of a certificate if i_π was initialized (Line 10). This is where it must be careful. The values in the pair $b_\pi = (b_\pi.pid, b_\pi.loc)$ are directly read from $B[i_\pi]$, and are thus potentially un-initialized ill-formed values. Thus, before the operation proceeds, it must check that $b_\pi.pid$ is indeed a real process id (Line 11). If so, it reads the control information in X$[b_\pi.pid]$ to get a pointer to $c_\pi^{other} = c_{b_\pi.pid}$ and the length of its valid sub-array x_π. *After checking that* $b_\pi.loc$ is indeed in the valid portion of c_π^{other}, it verifies that the location $c_\pi^{other}[b_\pi.loc]$ certifies $A[i_\pi]$ (Line 13). The careful well-formedness checks are necessary to avoid accessing un-allocated portions of memory. Of course, the operation returns *true* only if all of the checks pass (Line 13); if *any* checks fail (well-formedness or otherwise), it simply returns *false* (Line 14).

Remark 8.4.2 (old certification arrays)**.** A process π that is performing IsCertified$_\pi(i_\pi)$ may hold a reference c_π^{other} that is no longer the current array of any process. That is, after π reads $(x_\pi, c_\pi^{other}) \leftarrow$ X$[b_\pi.pid]$, the process $b_\pi.pid$ may have certified more elements and updated its current certification array. However, our algorithm remains correct, since the old certification arrays are not de-allocated, and the value of $c_\pi^{other}[b_\pi.loc]$ is guaranteed to be equal to $c_{b_\pi.pid}[b_\pi.loc]$—the corresponding value in the current array.

The preceding discussion is summarized in the theorem below.

Theorem 8.4.3. *Algorithm 24 is a linearizable wait-free fast array implementation for p processes. That is, it supports* Initialize$_\pi(m_\pi, f_\pi)$, Read$_\pi(i_\pi)$, *and* Write$_\pi(i_\pi, v_\pi)$ *each with a time complexity of $O(1)$. The total space complexity of the algorithm for supporting k fast-arrays of sizes $m_1, \ldots, m_k$ is $O(M + p)$, where $M = \sum_{j=1}^{k} m_j$.*

8.5 Correctness of Fast Array Algorithm

In this section, we will argue the correctness of Algorithm 24, for a fast-array object $\mathcal{O}$ of length m and initialization function $f_{init} = f$. In particular, we must show that the algorithm is linearizable and bounded wait-free. Bounded wait-freedom with a constant bound simply means that each of the operations runs in a constant number of steps—this is apparent from the pseudo-code. Thus, we immediately have the following lemma:

Lemma 8.5.1. *Algorithm 24 is bounded wait-free, and each operation runs in $O(1)$ time.*

The entire difficulty of the proof rests in showing linearizability of $\textsc{Read}_\pi()$ and $\textsc{Write}_\pi()$. As we have seen, the certification process is the workhorse of the algorithm. Analogously, formalizing and precisely stating the definition of the informal statement, "*index i is certified*" as a formal predicate $\textsc{Cert}(i)$ will enable the entire proof. We will define this formal predicate after writing down some quick definitions.

8.5.1 Some terminology

We will need the following definitions for our proof:

- For each index i, define α_i to be initial value of $A[i]$, and β_i to be the initial value of $B[i]$.
- Each $\mathrm{X}[\pi]$ holds a pair (x_π, c_π); we call the first index $\mathrm{X}[\pi].\mathit{valid\text{-}len}$ (because it is the length of the certification array that is valid) and the second index $\mathrm{X}[\pi].\mathit{arr}$ (because it points to the array).

8.5.2 Formalizing Certification: the essence of the proof

For a given index $i \in [m]$, we formalize the notion of index i being *certified*, by defining the predicate $\textsc{Cert}(i)$:

$$\textsc{Cert}(i) \equiv \exists \pi \in [p], j \in \mathbb{N}, (B[i] = (\pi, j) \neq \beta_i) \wedge (j < \mathrm{X}[\pi].\mathit{valid\text{-}len}) \wedge (c_\pi[j] = \&A[i])$$

Verbally, our definition means that an index is certified if three things hold true simultaneously: (1) $B[i]$ has changed from its initial value, (2) the certification location j in the array c_π that $B[i]$ indicates is valid, and (3) that certification location $c_\pi[j]$ indeed certifies index i. Notice that conjunct (1) implies that initially $\textsc{Cert}(i)$ is false for all indices $i \in [m]$. Notice that point number (1) is the principal difference between this definition and the analogous expression in the invariant $\mathcal{I}$ of the folklore algorithm.

We now prove a sequence of lemmas, whose end goal is to establish the correctness of $\textsc{Certify}(i)$ and $\textsc{IsCertified}(i)$. By this, we mean that any call to $\textsc{Certify}(i)$ will ensure that $\textsc{Cert}(i)$ becomes true; we will also show that $\textsc{Cert}(i)$ is monotonic, i.e., if it ever becomes true during

an execution, then it will stay true forever thereafter; finally, we will show that IsCertified(i) returns the value of Cert(i) (at some point during its execution).

Once we establish the correctness of the certification mechanism, the linearization points of Read() and Write() are easy to identify and prove correct.

8.5.3 The intermediate lemmas

Lemma 8.5.2. *Initially, Cert(i) is false for every index $i \in [m]$*

Proof. The first conjunct in Cert(i) states that $B[i] \neq \beta_i$. Since $\forall i \in [m], B[i] = \beta_i$ initially by definition, we have proved the lemma. □

Lemma 8.5.3. *For any given $i \in [m]$, if $B[i] \neq \beta_i$ at some point in time, then this implies that Cert(i). Furthermore, it implies that the value of $B[i] = (\pi, j)$, and the value of $c_\pi[j] = \&A[i]$ will never change in the future.*

Proof. We show this lemma by induction on the number of steps of the execution. As such, the base case is clear.

The crux of the inductive step is the following. A process π can only change the value of $B[i]$ by successfully executing the Cas on Line 31. π's Cas will only succeed if $B[i] = b_\pi$. Since b_π is read at the beginning of the Certify$_\pi(i)$ procedure on Line 16, and π will only get past Line 17 if b_π did not point to a certificate that certifies index i, so the inductive hypothesis would imply that $b_\pi = \beta_i$. So, there are only two cases: either (1) $b_\pi = \beta_i$ and the Cas will succeed, or (2) $b_\pi \neq \beta_i$ and the Cas will fail. Case (2) requires no further proof since it does not change the state of $B[i]$ and thus the inductive hypothesis directly implies the inductive step. In case (1), we must establish that if the Cas succeeds, then Cert(i) becomes true and that the values of $B[i] = (\pi, j)$ and $c_\pi[j] = \&A[i]$ that hold at this point in time will never change. For this, we observe that Lines 26-27 create a certificate for index i and increase X[π].*valid-len* so that the certificate is valid by the time $B[i] = (\pi, j)$ is established. Furthermore, since the Cas on Line 31 succeeds, walk-back will *not* take place on Line 32. Therefore, this certificate will never be altered while the current array remains c_π. Even when the current array gets updated to the next array, we know that $c'_\pi[0 \ldots c_\pi.len - 1] = c_\pi$ at the time that the next array becomes current. So, the certificate remains valid forever. Finally, $B[i]$ will not change in the future, since any process τ that reads $b_\tau \leftarrow B[i]$ on Line 16, will notice that the index i is already certified in the check on Line 17, and thereby will

not even attempt to change $B[i]$ on Line 31. We have finished the proof in both cases, and have thus established the lemma. □

The previous lemma establishes that for any given index i, the value of $B[i]$ changes at most once. Furthermore, if it does change, then the point of its change is precisely the point at which $\textsc{Cert}(i_\pi)$ goes from being *false* to being *true*, and this exact certificate will certify i forever on. Equipped with this lemma, we are ready to establish that the two helper methods are correct.

Lemma 8.5.4. *A call to $\textsc{IsCertified}_\pi(i_\pi)$ by process π returns the value of $\textsc{Cert}(i_\pi)$ at some time t during its execution. We call this time t the* actualization point *of the $\textsc{IsCertified}_\pi(i_\pi)$ operation.*

Proof. The first step by $\textsc{IsCertified}_\pi(i_\pi)$ is to read and store the value $b_\pi \leftarrow B[i_\pi]$ on Line 10. If $\textsc{Cert}(i_\pi)$ is true at this point in time, then $b_\pi = (\pi, j)$ must point to a valid certificate at the point in time that Line 10 is executed and all future times by Lemma 8.5.3. Thus, if π is able to verify that $c_\pi[j]$ is a valid certificate and indeed certifies index i, i.e. $c_\pi[j] = \&A[i_\pi]$, then it can validly return *true*; because, it must be the case that $\textsc{Cert}(i_\pi)$ is true at the time of execution of Line 10 by Lemma 8.5.3.

On the other hand, if one of the checks on Lines 11-13 fails, then we can surely conclude that $\textsc{Cert}(i_\pi)$ was false at the time of Line 10—by Lemma 8.5.3 and the definition of $\textsc{Cert}$).

So, regardless of what value $\textsc{IsCertified}_\pi(i_\pi)$ returns, we know it must be the value of $\textsc{Cert}(i_\pi)$ at some point during the methods execution. That concludes the proof. □

Lemma 8.5.5. *A call to $\textsc{Certify}_\pi(i_\pi)$ by process π will ensure that index i_π is certified before it returns.*

Proof. By Lemma 8.5.3 we must only establish that $\textsc{Cert}(i_\pi)$ is true at some point during the method.

The first step of $\textsc{Certify}_\pi(i_\pi)$ is to read the value $b_\pi \leftarrow B[i_\pi]$. If index i_π is already certified at this time, or if the call to $\textsc{IsCertified}_\pi(i_\pi)$ returns true on Line 17, then by Lemma 8.5.3, we conclude that $\textsc{Cert}(i_\pi)$ will continue to be true hereafter, by Lemma 8.5.4 we conclude that the method will return after Line 17.

Otherwise, we can conclude from Lemma 8.5.3 that $b_\pi = \beta_{i_\pi}$, when we start Line 18. In this case, the $\textsc{Certify}_\pi(i_\pi)$ method will continue and write and validate a certificate for index i at Lines 26–27, and attempt to place a pointer to this certificate at Line 31. If the $\textsc{Cas}$ on Line 31 succeeds,

we conclude that $\textsc{Cert}(i_\pi)$ must be true at that point and conclude by Lemma 8.5.3. Otherwise, we conclude that $B[i_\pi] \neq b_\pi = \beta_{i_\pi}$. This once again implies by Lemma 8.5.3 that $\textsc{Cert}(i_\pi)$ is true by the end of the method.

That concludes the proof. □

8.5.4 Linearizability of Read() and Write()

Now that we have established the workings of the certification mechanism, the linearization points of the $\textsc{Read}_\pi()$ and $\textsc{Write}_\pi()$ operations become apparent. The linearization points are as follows:

- A $\textsc{Write}_\pi(i_\pi, v_\pi)$ operation by process π linearizes at Line 4 if $\textsc{Cert}(i_\pi)$ is true at that time. Otherwise, it linearizes at the first time that $\textsc{Cert}(i_\pi)$ becomes true. The linearization points of all completed writes are well defined and within the invocation-response interval by Lemmas 8.5.5, which establishes that $\textsc{Cert}(i_\pi)$ becomes true before the end of Line 5. If several writes take effect at the same point in time, then we order them by the order in which they execute Line 4.

- A $\textsc{Read}_\pi(i_\pi)$ operation that returns $f_{init}(i_\pi)$ at Line 8 linearizes at the actualization point of its $\textsc{IsCertified}_\pi(i_\pi)$ call at Line 6. A $\textsc{Read}_\pi(i_\pi)$ that returns $A[i_\pi]$ at Line 7 linearizes at Line 7.

Lemma 8.5.6. *Algorithm 24 is linearizable, with $\textsc{Read}_\pi()$ and $\textsc{Write}_\pi()$ operations linearizing at the points presented in the itemized list above, and at all times $\forall i, \textsc{Cert}(i) \implies \mathcal{O}[i] = A[i]$ according to the claimed linearization.*

Proof. The itemized list justifies that the claimed linearization points exist and occur within the invocation-response intervals of the given operations. The remainder of the proof is by induction over the number of linearization points. The base case is trivially true, and the inductive hypothesis assumes that all previous linearization points are consistent.

The linearization points of the $\textsc{Write}(i, v)$ operations are justified because they maintain the invariant that $\textsc{Cert}(i) \implies \mathcal{O}[i] = A[i]$ that we claim in the theorem statement.

By the given linearization points, no $\textsc{Write}(i)$ operation linearizes before $\textsc{Cert}(i)$ becomes true. Thus, a $\textsc{Read}(i)$ that returns $f_{init}(i)$ is justified in linearizing at the actualization point of the $\textsc{IsCertified}(i)$ call on Line 6, since $\textsc{Cert}(i) = \mathit{false}$ at that point by Lemma 8.5.4. Similarly, a $\textsc{Read}(i)$ that returns $A[i]$ on Line 7 is justified in linearizing at the same line, since $A[i] = \mathcal{O}[i]$ once i_π is initialized by the linearization points of previous write operations. □

Lemma 8.5.1 and Lemma 8.5.6 together establish the correctness of the main theorem about our fast-array Algorithm 24. Thus we have justified Theorem 8.4.3.

8.6 A Concurrent Fast Generalized Array

In this section, we implement fast generalized arrays, which we motivated in Section 8.1.2.

Recall that, if $\mathcal{S}$ is the set of hardware-supported RMW primitives, then a *fast generalized array* is an implementation that not only supports $O(1)$-time linearizable $\textsc{Initialize}(m, f)$, $\textsc{Read}(i)$, and $\textsc{Write}(i, v)$ operations, but also supports $O(1)$-time linearizable operations from the set $\mathcal{S}$. To this end, we consider the operation:

- $\mathcal{O}.\textsc{Apply}(i, op, args)$: perform operation op with arguments $args$ on $\mathcal{O}[i]$, and return the response.

Here op can be any RMW operation—such as Write, CAS, FAA, or FAS—that is supported in hardware, and $args$ are the arguments that the primitive requires. For example, if $\mathcal{O}[5] = 17$, then a call to $\mathcal{O}.\textsc{Apply}(5, \textsc{Cas}, (17, 35))$ changes the value of $\mathcal{O}[5]$ to 35 and returns *true*. We term an array that supports $\textsc{Initialize}(m, f)$, $\textsc{Read}(i)$, and $\textsc{Apply}(i, op, args)$, a *generalized array*, and an implementation that runs each operation in $O(1)$ time, a *fast generalized array*. Note that a $\textsc{Write}(i, v)$ can be executed as $\textsc{Apply}(i, \textsc{Write}, v)$. While $\textsc{Read}(i)$ can similarly be executed using $\textsc{Apply}(i, \textsc{Read})$, we design a simpler read method that circumvents the certification overhead for locations that are only read and never updated.

The goal of this section is to design a fast generalized array. We will achieve this goal by building on our ideas from Algorithm 24 for concurrent fast arrays. Therefore, we will continue to use the ideas of individual certification arrays, synchronization and walk-back, array doubling, sharing of certification arrays, and tombstoning. Even so, supporting arbitrary RMW operations poses yet new challenges. We first describe these challenges, and then explain how we overcome them.

Recall that at a high level, our fast array algorithm represents each abstract array element $\mathcal{O}[i]$ by the value of $A[i]$, along with the certification mechanism which keeps track of whether i has been initialized. Thus, $\textsc{Read}(i)$ simply returns $A[i]$ if i is initialized and $f(i)$ otherwise. Write operations, on the other hand, follow an *apply-then-certify* scheme. That is, $\textsc{Write}(i, v)$ blindly *applies* its operation by writing $A[i] \leftarrow v$, and subsequently *certifies* i if necessary.

A natural idea for implementing an RMW operation on $\mathcal{O}[i]$ would be to mimic the apply-then-certify scheme used by writes in Algorithm 24. For example, $\mathcal{O}.\textsc{Apply}(i, \textsc{Cas}, (old, new))$ would

blindly apply $r \leftarrow \textsc{Cas}(A[i], old, new)$, and then certify i if necessary, and finally return r. Indeed, this idea would work if i were already initialized, since, in that case, $A[i]$ would hold the value of $\mathcal{O}[i]$. However, if i were not already initialized, then the abstract element $\mathcal{O}[i]$ has the value $f(i)$, while $A[i]$ has some arbitrary value. In particular, if $f(i) = old$, but $A[i] = some\text{-}other\text{-}value$ (not equal to old), then $\mathcal{O}.\textsc{Apply}(i, \textsc{Cas}, (old, new))$ should change $\mathcal{O}[i]$'s value to new and return *true*, but the proposed scheme would keep the value the same (at $\mathcal{O}[i] = A[i] = some\text{-}other\text{-}value$), certify index i, and return *false*. Thus, both the final value of $\mathcal{O}[i]$ and the return value of the operation would be incorrect.

From the example above, we see that RMW operations are difficult to apply before index i is initialized and certified, but easy to apply after the certification process. So, our idea is to reverse the scheme, rather than *apply-then-certify*, we will implement *certify-then-apply*. Since this new scheme will ensure that i is always certified first, the actual application of the RMW primitive can be realized as a hardware primitive applied directly to $A[i]$. Consequently, we can apply *any* primitive operation that hardware supports, not just the select few that we listed at the beginning of the section.

Certifying first poses a new challenge. When we applied writes to $A[i]$ before certifying, we were guaranteed that $A[i]$ would hold a valid (linearizable) value by the time i was certified. Since a reader will return $A[i]$ as the value of $\mathcal{O}[i]$ any time after i is certified, we still need to guarantee that $A[i] = \mathcal{O}[i]$ at the time of certification. This seems to be a difficult requirement with our current setup, since our previous certification process did not touch $A[i]$, but rather linearized at the time that a Cas was performed on $B[i]$. To overcome this challenge, we introduce the idea of *fusing* as described below.

Fusing: The values stored in each $A[i]$ correspond to the values stored in the corresponding abstract element $\mathcal{O}[i]$. So, it is important to allow these values to take up a full-pointer sized word, e.g., a 64-bit full-word in a modern 64-bit architecture. The pairs (π, j) that we are storing in $B[i]$ however, are just an internal representation used by our algorithm. Furthermore, it is entirely reasonable to assume that this pair can be stored in a single full-word. For example, allocating 14-bits for the process id π would allow for over 16,000 processors, and the remaining 50-bits would be enough to index an array with a thousand-trillion indices (i.e., an array taking up 8000 terabytes of memory). Therefore, we modify our representation by, intuitively, "absorbing the array B into A". Now, each element of our array $A[i]$ will hold a triple $(A[i].val, A[i].pid, A[i].loc)$, where the *value* $A[i].val$ is stored in the first word and the *process id* $A[i].pid$ and the *location* $A[i].loc$ are

Algorithm 25 Atomic fast generalized array for p processes. Pseudo-code shown for an arbitrary process π.

Variables:

For each process $\pi \in [p]$ the following variables are shared across *all* fast-arrays $\mathcal{O}$:

- $c_\pi[0,1]$ is a pointer to an allocated un-initialized array of length 2.
- $c'_\pi[0,\ldots,3]$ is a pointer to an allocated un-initialized array of length 4.
- k_π is a non-negative integer that is initialized to 0.
- X is an array, where each X$[\pi]$ stores pair that is initialized to $(0, c_\pi)$.

Each object $\mathcal{O}$ has two instance variables instantiated by INITIALIZE$_\pi(m_\pi, f_\pi)$:

- A is an array of double words.
- f_{init} stores the initial value function.

Each process $\pi \in [p]$ uses the following arbitrarily initialized temporary local variables:

- a_π, a_π^{old}: hold (value, process id, array index) triples.
- x_π: holds an array index.
- c_π^{other}: holds an array pointer.

procedure $\mathcal{O}$.INITIALIZE$_\pi(m_\pi, f_\pi)$
1: $A \leftarrow$ **new** double-width-array$[m_\pi]$
2: $f_{init} \leftarrow f_\pi$

procedure $\mathcal{O}$.APPLY$_\pi(i_\pi, op_\pi, args_\pi)$
3: CERTIFY$_\pi(i_\pi)$
4: **return** $op_\pi(A[i_\pi].val, args_\pi)$

procedure $\mathcal{O}$.READ$_\pi(i_\pi)$
5: **if** ISCERTIFIED$_\pi(i_\pi)$ **then return** $A[i_\pi].val$ **else return** $f_{init}(i_\pi)$

6: **procedure** $\mathcal{O}$.ISCERTIFIED$_\pi(i_\pi)$
7: $a_\pi \leftarrow A[i_\pi]$
8: **if** $0 \le a_\pi.pid < p$ **then**
9: $(x_\pi, c_\pi^{other}) \leftarrow$ X$[a_\pi.pid]$
10: **if** $0 \le a_\pi.loc < x_\pi$ **and** $c_p^{other}[a_\pi.loc] = \&A[i_\pi]$ **then return** *true*
11: **return** *false*

12: **procedure** $\mathcal{O}$.CERTIFY$_\pi(i_\pi)$
13: $a_\pi^{old} \leftarrow A[i_\pi]$
14: **if** ISCERTIFIED$_\pi(i_\pi)$ **then return**
15: $(x_\pi, -) \leftarrow$ X$[\pi]$
16: **if** $x_\pi \ge c_\pi.len$ **then**
17: $c_\pi \leftarrow c'_\pi$
18: $c'_\pi \leftarrow$ **new** array$[2 \cdot c_\pi.len]$
19: $k_\pi \leftarrow 0$
20: **if** $a_\pi^{old}.pid = p$ **and** $a_\pi^{old}.loc = x_\pi$ **then**
21: $c_\pi[x_\pi] = \perp$
22: $x_\pi \leftarrow x_\pi + 1$
23: $c_\pi[x_\pi] \leftarrow \&A[i_\pi]$
24: X$[\pi] \leftarrow (x_\pi + 1, c_\pi)$
25: **while** $k_\pi < 2x_\pi - c_\pi.len$ **do**
26: $c'_\pi[k_\pi] \leftarrow c_\pi[k_\pi]$
27: $k_\pi \leftarrow k_\pi + 1$
28: **if not** CAS$(A[i_\pi], a_\pi^{old}, (f_{init}(i_\pi), \pi, x_\pi))$ **then**
29: X$[\pi] \leftarrow (x_\pi, c_\pi)$

packed into the second word of a *double-width word*. Modern architectures, such as x86-64, allow us to perform double-width CAS operations on the full double-word $A[i]$, while also allowing all the standard single-width hardware primitives (CAS, FAA, FAS, WRITE, etc.) on the first word $A[i].val$. Using this feature of hardware, we can safely implement the "certify" portion of the certify-then-apply scheme. In particular, if process π reads $a_0 \leftarrow A[i]$ in its "un-initialized" state, and creates a certificate for it in $c_\pi[j]$, it can perform the certify step via: $\text{CAS}(A[i], a_0, (f(i), \pi, j))$.

8.6.1 The pseudo-code and its description

The pseudo-code for a process π's operations on our fast generalized array is presented as Algorithm 25. The algorithm is built on all of the ideas from the previous section—individual certification arrays, synchronization and walk-back, concurrent array-doubling, tombstoning, and certification mechanism sharing—along with the ideas introduced above—fusing, and the certify-then-apply scheme. We proceed to briefly describe the pseudo-code below.

The code of the three operations in the interface is simple to understand. $\mathcal{O}.\text{INITIALIZE}_\pi(m_\pi, f_\pi)$ simply instantiates a single new un-initialized array A (Line 1), and stores the initialization function (Line 2). $\mathcal{O}.\text{APPLY}_\pi(i_\pi, op_\pi, args_\pi)$ executes certify-then-apply by simply certifying at Line 3 and applying (and returning) at Line 4. $\mathcal{O}.\text{READ}_\pi(i_\pi)$ simply returns $A[i_\pi]$'s value field val if i_π is certified and $f_{init}(i_\pi)$ otherwise (Line 5).

Once again, the main workhorse of the algorithm is the certification mechanism. $\mathcal{O}.\text{CERTIFY}_\pi(i_\pi)$ returns early if i_π is already certified (Lines 13–14). Otherwise, it loads the next available certification location x_π from $\text{X}[\pi]$ at Line 15, and follows the same logical steps as our earlier certification method: (1) update current arrays if necessary (Lines 16–19), (2) tombstone the location if the nasty race condition might arise (Lines 20–22), (3) create a certificate for $A[i_\pi]$ (Lines 23–24), (4) transfer values to the next array (Lines 25–27), and (5) synchronize, and walk-back if necessary (Lines 28–29). The most noteworthy difference from Algorithm 24 is Line 28, where we perform the double-width CAS operation to simultaneously update $A[i_\pi].val$ to $f_{init}(i_\pi)$ and $(A[i_\pi].pid, A[i_\pi].loc)$ to (π, x_π).

$\mathcal{O}.\text{ISCERTIFIED}_\pi(i_\pi)$ now reads a triple a_π (rather than the pair b_π) at Line 7, but performs the same logical function as in the standard fast array. It returns *true* only if $a_\pi.pid$ and $c_{a_\pi.pid}[a_\pi.loc]$ are valid, and if so certifies $A[i_\pi]$ (Lines 8–10). Otherwise, it returns *false* (Line 11).

The preceding discussion is summarized in the theorem below.

Theorem 8.6.1. *Algorithm 25 is a linearizable wait-free fast generalized array implementation for p processes. That is, for each process $\pi \in [p]$, it supports* $\textsc{Initialize}_\pi(m_\pi, f_\pi)$, $\textsc{Read}_\pi(i_\pi)$, *and* $\textsc{Apply}_\pi(i_\pi, op_\pi, args_\pi)$ *each with a time complexity of $O(1)$. The total space complexity of the algorithm for supporting k fast generalized arrays of sizes $m_1, \ldots, m_k$ is $O(M + p)$, where $M = \sum_{j=1}^{k} m_j$, given that each memory word has at least $\log_2 M + \log_2 p$ bits.*

8.7 Correctness of Fast Generalized Array Algorithm

In this section, we will argue the correctness of Algorithm 25, for a fast generalized array object $\mathcal{O}$ of length m and initialization function $f_{init} = f$. In particular, we must show that the algorithm is linearizable and bounded wait-free. Bounded wait-freedom with a constant bound simply means that each of the operations runs in a constant number of steps—this is apparent from the pseudo-code. Thus, we immediately have the following lemma:

Lemma 8.7.1. *Algorithm 25 is bounded wait-free, and each operation runs in $O(1)$ time.*

The entire difficulty of the proof rests in showing linearizability of $\textsc{Read}_\pi()$ and $\textsc{Apply}_\pi()$. As we have seen, the certification process is the workhorse of the algorithm. Analogously, formalizing and precisely stating the definition of the informal statement, "*index i is certified*" as a formal predicate $\textsc{Cert}(i)$ will enable the entire proof. We will define this formal predicate after writing down some quick definitions.

8.7.1 Some terminology

We will need the following definitions for our proof:

- For each index i, define α_i to be initial value of $A[i]$.
- Each $\mathrm{X}[\pi]$ holds a pair (x_π, c_π); we call the first index $\mathrm{X}[\pi].$*valid-len* (because it is the length of the certification array that is valid) and the second index $\mathrm{X}[\pi].arr$ (because it points to the array).

8.7.2 Formalizing Certification: the essence of the proof

For a given index $i \in [m]$, we formalize the notion of index i being *certified*, by defining the predicate $\textsc{Cert}(i)$:

$$\begin{aligned}\textsc{Cert}(i) \equiv \exists \pi \in [p], j \in \mathbb{N},& \\ & ((A[i].pid, A[i].loc) = (\pi, j) \neq (\alpha_i.pid, \alpha_i.loc)) \\ & \wedge \ (j < \mathrm{X}[\pi].\textit{valid-len}) \\ & \wedge \ (c_\pi[j] = \&A[i].val)\end{aligned}$$

Verbally, our definition means that an index is certified if three things hold true simultaneously: (1) the $(A[i].pid, A[i].loc)$ has changed from its initial value, (2) the certification location j in the array c_π that $(A[i].pid, A[i].loc)$ indicates is valid, and (3) that certification location $c_\pi[j]$ indeed certifies index i. Notice that conjunct (1) implies that initially $\textsc{Cert}(i)$ is false for all indices $i \in [m]$.

We now prove a sequence of lemmas, whose end goal is to establish the correctness of $\textsc{Certify}(i)$ and $\textsc{IsCertified}(i)$. By this, we mean that any call to $\textsc{Certify}(i)$ will ensure that $\textsc{Cert}(i)$ becomes true; we will also show that $\textsc{Cert}(i)$ is monotonic, i.e., if it ever becomes true during an execution, then it will stay true forever thereafter; finally, we will show that $\textsc{IsCertified}(i)$ returns the value of $\textsc{Cert}(i)$ (at some point during its execution).

Once we establish the correctness of the certification mechanism, the linearization points of $\textsc{Read}()$ and $\textsc{Apply}()$ are easy to identify and prove correct.

8.7.3 The intermediate lemmas

Lemma 8.7.2. *Initially, $\textsc{Cert}(i)$ is false for every index $i \in [m]$*

Proof. The first conjunct in $\textsc{Cert}(i)$ states that $(A[i].pid, A[i].loc) \neq (\alpha_i.pid, \alpha_i.loc)$. Since $\forall i \in [m], (A[i].pid, A[i].loc) = (\alpha_i.pid, \alpha_i.loc)$ initially by definition, we have proved the lemma. □

Lemma 8.7.3. *For any given $i \in [m]$, if $(A[i].pid, A[i].loc) \neq (\alpha_i.pid, \alpha_i.loc)$ at some point in time, then this implies that $\textsc{Cert}(i)$. Furthermore, it implies that the value of $(A[i].pid, A[i].loc) = (\pi, j)$, and the value of $c_\pi[j] = \&A[i]$ will never change in the future.*

Proof. We show this lemma by induction on the number of steps of the execution. As such, the base case is clear.

The crux of the inductive step is the following. A process π can only change the value of $(A[i].pid, A[i].loc)$ by successfully executing the $\textsc{Cas}$ on Line 28. π's $\textsc{Cas}$ will only succeed if

$A[i] = a_\pi$. Since a_π is read at the beginning of the CERTIFY$_\pi(i)$ procedure on Line 13, and π will only get past Line 14 if b_π did not point to a certificate that certifies index i, so the inductive hypothesis would imply that $(A[i].pid, A[i].loc) = (\alpha_i.pid, \alpha_i.loc)$. So, there are only two cases: either (1) $a_\pi = \alpha_i$ and the CAS will succeed, or (2) $a_\pi \neq \alpha_i$ and the CAS will fail. Case (2) requires no further proof since it does not change the state of $A[i]$ and thus the inductive hypothesis directly implies the inductive step. In case (1), we must establish that if the CAS succeeds, then CERT(i) becomes true and that the values of $(A[i].pid, A[i].loc) = (\alpha_i.pid, \alpha_i.loc)$ and $c_\pi[j] = \&A[i]$ that hold at this point in time will never change. For this, we observe that Lines 23–24 create a certificate for index i and increase X$[\pi]$.*valid-len* so that the certificate is valid by the time $(A[i].pid, A[i].loc) = (\pi, j)$ is established. Furthermore, since the CAS on Line 28 succeeds, walk-back will *not* take place on Line 29. Therefore, this certificate will never be altered while the current array remains c_π. Even when the current array gets updated to the next array, we know that $c'_\pi[0 \ldots c_\pi.len - 1] = c_\pi$ at the time that the next array becomes current. So, the certificate remains valid forever. Finally, $(A[i].pid, A[i].loc)$ will not change in the future, since any process τ that reads $a_\tau \leftarrow A[i]$ on Line 13, will notice that the index i is already certified in the check on Line 14, and thereby will not even attempt to change $A[i]$ on Line 28. We have finished the proof in both cases, and have thus established the lemma. □

The previous lemma establishes that for any given index i, the value of $(A[i].pid, A[i].loc)$ changes at most once. Furthermore, if it does change, then the point of its change is precisely the point at which CERT(i) goes from being *false* to being *true*, and this exact certificate will certify i forever on. Equipped with this lemma, we are ready to establish that the two helper methods are correct.

Lemma 8.7.4. *A call to* IsCERTIFIED$_\pi(i_\pi)$ *by process* π *returns the value of* CERT(i_π) *at some time* t *during its execution. We call this time* t *the* actualization point *of the* IsCERTIFIED$_\pi(i_\pi)$ *operation.*

Proof. The first step by IsCERTIFIED$_\pi(i_\pi)$ is to read and store the value $a_\pi \leftarrow A[i_\pi]$ at Line 7. If CERT(i_π) is true at this point in time, then $(A[i].pid, A[i].loc) = (\pi, j)$ must point to a valid certificate at the point in time that Line 10 is executed and all future times by Lemma 8.5.3. Thus, if π is able to verify that $c_\pi[j]$ is a valid certificate and indeed certifies index i, i.e. $c_\pi[j] = \&A[i_\pi]$, then it can validly return *true*; because, it must be the case that CERT(i) is true at the time of execution of Line 10 by Lemma 8.5.3.

On the other hand, if one of the checks on Lines 8-10 fails, then we can surely conclude that $\textsc{Cert}(i_\pi)$ was false at the time of Line 7—by Lemma 8.5.3 and the definition of $\textsc{Cert}$).

So, regardless of what value $\textsc{IsCertified}_\pi(i_\pi)$ returns, we know it must be the value of $\textsc{Cert}(i_\pi)$ at some point during the methods execution. That concludes the proof. □

Lemma 8.7.5. *A call to $\textsc{Certify}_\pi(i_\pi)$ by process π will ensure that index i_π is certified before it returns.*

Proof. By Lemma 8.7.3 we must only establish that $\textsc{Cert}(i_\pi)$ is true at some point during the method.

The first step of $\textsc{Certify}_\pi(i_\pi)$ is to read the value $a_\pi \leftarrow A[i_\pi]$. If index i_π is already certified at this time, or if the call to $\textsc{IsCertified}_\pi(i_\pi)$ returns true on Line 14, then by Lemma 8.7.3, we conclude that $\textsc{Cert}(i_\pi)$ will continue to be true hereafter, by Lemma 8.7.4 we conclude that the method will return after Line 14.

Otherwise, we can conclude from Lemma 8.7.3 that $b_\pi = \beta_i$, when we start Line 15. In this case, the $\textsc{Certify}_\pi(i_\pi)$ method will continue and write and validate a certificate for index i at Lines 23–24, and attempt to place a pointer to this certificate at Line 28. If the $\textsc{Cas}$ on Line 28 succeeds, we conclude that $\textsc{Cert}(i_\pi)$ must be true at that point and conclude by Lemma 8.7.3. Otherwise, we conclude that $A[i] \neq a_\pi = \alpha_i$. This once again implies by Lemma 8.7.3 that $\textsc{Cert}(i_\pi)$ is true by the end of the method.

That concludes the proof. □

8.7.4 Linearizability of Read() and Apply()

Now that we have established the workings of the certification mechanism, the linearization points of the $\textsc{Read}_\pi()$ and $\textsc{Apply}_\pi()$ operations become apparent. The linearization points are as follows:

- An $\textsc{Apply}_\pi(i_\pi, op_\pi, args_\pi)$ operation linearizes at Line 4.
- A $\textsc{Read}_\pi(i_\pi)$ operation that returns $f_{init}(i_\pi)$ at the actualization point of its $\textsc{IsCertified}_\pi(i_\pi)$ call at Line 5. A $\textsc{Read}_\pi(i_\pi)$ that returns $A[i_\pi]$ linearizes exactly when it returns.

Lemma 8.7.6. *Algorithm 25 is linearizable, with $\textsc{Read}_\pi()$ and $\textsc{Apply}_\pi()$ operations linearizing at the points presented in the itemized list above, and at all times $\forall i, \textsc{Cert}(i) \implies \mathcal{O}[i] = A[i].val$ according to the claimed linearization.*

Proof. The itemized list justifies that the claimed linearization points exist and occur within the invocation-response intervals of the given operations. The remainder of the proof is by induction over the number of linearization points. The base case is trivially true, and the inductive hypothesis assumes that all previous linearization points are consistent.

The linearization points of the $\textsc{Apply}(i, op, args)$ operations are justified because they maintain the invariant that $\textsc{Cert}(i) \implies \mathcal{O}[i] = A[i]$ that we claim in the theorem statement. And return the result of calling the operation op with arguments $args$ on $\mathcal{O}[i] = A[i]$, by making the call and return on Line 4.

By the given linearization points, no $\textsc{Apply}(i)$ operation linearizes before $\textsc{Cert}(i)$ becomes true. Thus, a $\textsc{Read}(i)$ that returns $f_{init}(i)$ is justified in linearizing at the actualization point of the $\textsc{IsCertified}(i)$ call, since $\textsc{Cert}(i) = \mathit{false}$ at that point by Lemma 8.7.4. Similarly, a $\textsc{Read}(i)$ that returns $A[i]$ on Line 5 is justified in linearizing at the same time, since $A[i] = \mathcal{O}[i]$ once i is certified by the linearization points of previous apply operations. □

Lemma 8.7.1 and Lemma 8.7.6 together establish the correctness of the main theorem about our fast-array Algorithm 25. Thus we have justified Theorem 8.6.1.

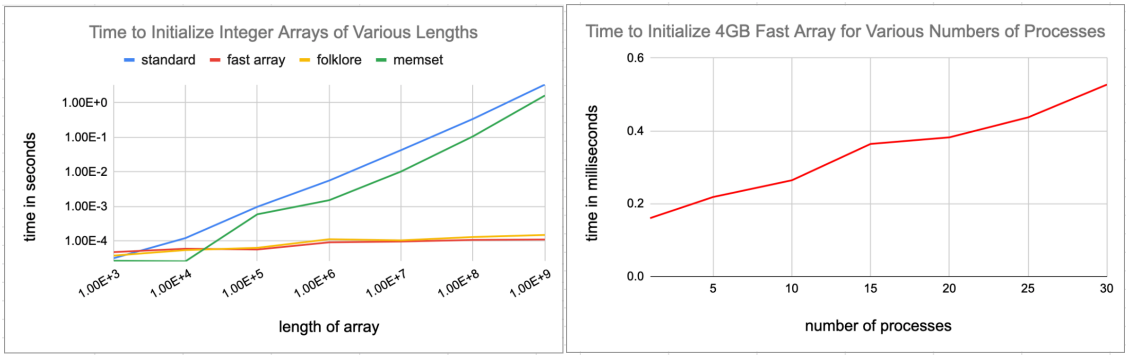
8.8 Experiments

We perform experiments using two 8-core Intel Xeon E5-2670 CPUs with two-way hyper-threading. The machine has 64GB of DRAM. Our machine ran 64-bit Ubuntu 12.04 with Linux kernel 3.13.0-143. All algorithms were coded in C++ and without any optimizations or specific algorithmic engineering to increase the speeds from the pseudo-code presented. We used `std::threads` to implement our concurrent fast array, and we compiled our code with g++ version 4.8.4 with the `-std=c++11`, `-pthread`, and `-mcx16` options set. We experiment with four different algorithms:

1. *standard:* a classic array, where initialization is performed by a linear-time for-loop through the indices of the array.

2. *memset:* a classic array, where initialization is performed by the C++ `memset` primitive. The `memset` operation can only be used to initialize an array to all 0s. In particular, it cannot be used to initialize it to an arbitrary function f.

3. *folklore:* the sequential folklore fast array algorithm (i.e., Algorithm 23). This algorithm can only be used by a single process; it is not a concurrent algorithm, but it can serve as a



(a) (b)

Figure 8.8.1: Figure 8.8.1a is a log-log plot charting the time to initialize arrays of various lengths for a single process. Figure 8.8.1b is a plot charting the times to initialize fast arrays of length one billion for various numbers of processes.

baseline.

4. *fast array:* our concurrent fast array (i.e., Algorithm 24).

Our experiments focus on measuring the speeds of the three operations—INITIALIZE(), READ(), and WRITE(). We present the results below.

1. To compare the speed of INITIALIZE() across all four algorithms, we measure the time to initialize arrays to all zeroes. The array lengths we test are $m = 10^k$ for $k \in \{3, 4, 5, 6, 7, 8, 9\}$ with each entry being 4 bytes, i.e., arrays of size 4KB to 4GB. As predicted by the algorithmic analysis, the folklore and fast array algorithms take only constant time to initialize, while initializing by for-loop and `memset` take linear time.

   As shown in Figure 8.8.1a, initializing an array with `memset` is 1.2–4.7 times faster than initializing with a for-loop, however initializing a fast array of length one billion is more than 14,000 times faster than initializing with `memset`. As the length of the array gets smaller, the initialization time advantage of fast arrays reduces. Fast array initialization and `memset` initialization become equally fast at an array length between 10^4 and 10^5, and fast array initialization and for-loop initialization become equally fast at an array length between 10^3 and 10^4. We consistently observe that initializing a fast array takes only as much time as initializing a folklore array. As shown in Figure 8.8.1b, it takes only 3.3 times more time to initialize a 4GB fast array for 30 processes rather than one for a single process.

2. While fast arrays are much faster to initialize than standard arrays, read and write operations are slower on these arrays at all levels of concurrency. In order to measure exactly how much

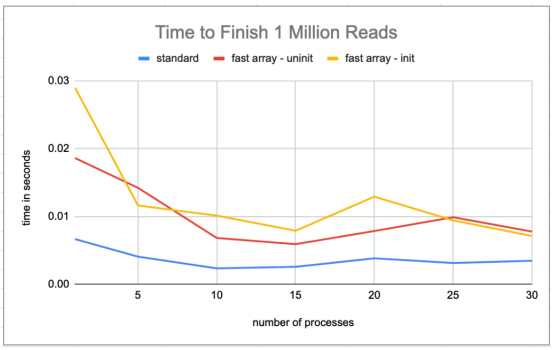


Figure 8.8.2: This plot compares the time to read from one million indices of a standard array versus a fast array for various numbers of concurrent processes. For the fast array, both reads from uninitialized and initialized indices are compared.

slower fast array reads are, we measure the cumulative time to perform one million reads using each type of array. Since the fast array READ() algorithm is different for indices that have been "initialized"—i.e., written to at least once after initialization—versus those that are "uninitialized", we measure the two types of reads separately (see Figure 8.8.2).

The main takeaways of the experiment are as follows: (1) the two different types of reads—initialized and uninitialized—on fast arrays are of comparable speed; (2) fast array reads are 2.1–4.3 times slower than reads to standard arrays.

3. Algorithm 24 suggests that writes to "initialized" indices should be faster than those to "un-initialized" indices. This is also confirmed by our experiment shown in Figure 8.8.3. In particular, fast array writes to un-initialized indices are 6–21 times slower than standard writes, but writes to initialized indices are only 2.2–4.9 times slower. It is noteworthy here that the slower speed only occurs once per index, and all subsequent writes to that index happen at the faster speed.

4. Writing to a new index in a fast array is 6–21× slower than writing to a new index in a naive array. So, fast arrays maintain their initial advantage over standard arrays as long as only 4–17% of the array indices are written to.

8.9 Application: Fixed Size Hash Table

To illustrate the power of the fast arrays, we present an implementation of a *provably efficient* fixed-size concurrent hash table built using fast generalized array as Algorithm 26.

Figure 8.8.3: This plot compares the time to write to one million indices of a standard array versus a fast array for various numbers of concurrent processes. For the fast array, both writes to uninitialized and initialized indices are compared.

8.10 Algorithm

Theorem 8.10.1. *Let H be a fixed size hash table, whose code is described by Algorithm 26. Then, the worst-case time complexity of* INITIALIZE$_\pi$ *is $O(1)$ if the* ARRAY *is a fast-generalized-array and $O(m_\pi)$ if the array is a standard array. With an oblivious scheduler, the expected time complexity of each of* INSERT$_\pi$ *and* GET$_\pi$ *is $O(1)$ if the total number of different keys inserted into H over the history is at most λm_π for any fixed constant* load factor $\lambda < 1$.

Proof. Let $n \leq \lambda m$ (we will use the notation $m = m_\pi$ for simplicity hereafter), be the total number of distinct keys inserted into the hash table over the entire history of the object. Once a key k is inserted, k's position in the array is unique and unchanging over time, i.e. there is exactly one position in the array that has the key k and that position never changes. Thus, we can consider the state of the array $A = H.A$ after all the distinct keys have been inserted. We consider an array component $A[i]$ *occupied* if $A[i]$.EXIT $\neq \perp$, and we consider it *empty* otherwise. Thus, once the n keys have been inserted into A, the array will have certain contiguous sequences of array components that are occupied that are preceded and succeeded by empty components; we call these contiguous sequences *chunks*. For the purposes of chunks, we consider A to be circular and all array indices we refer to in the analysis are mod m. We define the (probabilistic) event CHUNK(i, ℓ) to mean that there is a chunk of *length* ℓ starting at index i, i.e., $A[i], \ldots, A[i+\ell-1]$ are occupied and both $A[i-1]$ and $A[i+\ell]$ are empty.

Now consider the first time (by linearization) that a key k is inserted into H. k will occupy a new location in A during this insertion, and the time taken by the insertion will be at most the length of the chuck that contains the index $h(k)$ (if there is such a chunk) plus $O(1)$. Thus, using

Algorithm 26 Fixed size linear probing hash table using fast arrays.

```
    procedure H ← INITIALIZE_π(m_π, h_π)    ▷ m_π is the size of the table. h_π is the hash function.
 1:     A ← new ARRAY(m_π, (⊥, ⊥))
 2:     h ← h_π

    procedure H.INSERT_π(k_π, v_π)
 3:     i_π ← h(k_π)
 4:     while true do
 5:         x_π ← A[i_π]
 6:         if x_π.EXIT = k_π then CAS(A[i_π], x_π, (k_π, v_π)); return
 7:         else if x_π.EXIT = ⊥ then
 8:             if CAS(A[i_π], x_π, (k_π, v_π)) then return
 9:             else if A[i_π].EXIT = k_π then return
10:         i_π ← ((i_π + 1) mod A.len)

    procedure H.GET_π(k_π)
11:     i_π ← h(k_π)
12:     while true do
13:         x_π ← A[i_π]
14:         if x_π.EXIT = k_π then return x_π.val
15:         else if x_π.EXIT = ⊥ then return ⊥
16:         else i_π ← ((i_π + 1) mod A.len)

    procedure H.DELETE_π(k_π)
17:     INSERT_π(k_π, ⊥)
```

$\mathbb{1}(prop)$ to refer to the indicator variable of proposition *prop*, we see that the time, $T(k)$, taken to insert k is bounded by:

$$T(k) \leq O(1) + \sum_{i=1}^{m} \sum_{\ell=1}^{n} \ell \cdot \mathbb{1}(\textsc{Chunk}(i, \ell) \wedge h(k) \in [i, i + \ell))$$

Given that the chunk $[i, i + \ell)$ exists, the probability of a new (never before inserted) key k hashing into the chunk is ℓ/m. Thus, we can bound the expected insertion time as follows:

$$
\begin{aligned}
\mathbb{E}_h T(k) &\leq O(1) + \mathbb{E}_h \sum_{i=1}^{m} \sum_{\ell=1}^{n} \ell \cdot \mathbb{1}(\textsc{Chunk}(i,\ell) \wedge h(k) \in [i, i+\ell)) \\
&= O(1) + \sum_{i=1}^{m} \sum_{\ell=1}^{n} \ell \cdot \mathbb{E}_h \mathbb{1}(\textsc{Chunk}(i,\ell) \wedge h(k) \in [i, i+\ell)) \\
&= O(1) + \sum_{i=1}^{m} \sum_{\ell=1}^{n} \ell \cdot P_h(\textsc{Chunk}(i,\ell) \wedge h(k) \in [i, i+\ell)) \\
&= O(1) + \sum_{i=1}^{m} \sum_{\ell=1}^{n} \ell \cdot P_h(\textsc{Chunk}(i,\ell)) \cdot P_h(h(k) \in [i, i+\ell) \mid \textsc{Chunk}(i,\ell)) \\
&= O(1) + \sum_{i=1}^{m} \sum_{\ell=1}^{n} \ell \cdot P_h(\textsc{Chunk}(i,\ell)) \cdot \frac{\ell}{m} \\
&= O(1) + \sum_{\ell=1}^{n} \ell^2 \cdot P_h(\textsc{Chunk}(i,\ell))
\end{aligned}
$$

In order to bound the terms of the form $P(\textsc{Chunk}(i,\ell))$, we observe that $\textsc{Chunk}(i,\ell)$ implies that at least ℓ of the n keys hash to values in the range $[i, i+\ell)$. Thus, defining $\textsc{Hits}(i,\ell)$ to be the event that at least ℓ of the keys hash into the range $[i, i+\ell)$, we observe

$$
\mathbb{E}_h T(k) \leq O(1) + \sum_{\ell=1}^{n} \ell^2 \cdot P_h(\textsc{Hits}(i,\ell))
$$

Here, we observe that by the uniformity of the hash value of a given key and the mutual independence of the hash values of the n distinct keys, the random variables $X = \sum_{i=1}^{n} X_i$ where each X_i indicates whether the i^{th} key hashes to the range $[i, i+\ell)$ is a sum of independent indicators. Thus, $P_h(\textsc{Hits}(i,\ell)) = P_h(X \geq \ell)$. Since $\mathbb{E}_h X = \sum_{i=1}^{n} \mathbb{E}_h X_i = \frac{n\ell}{m} \leq \lambda\ell$. We use Chernoff's bound (Theorem 3.2 from [35]) to evaluate that:

$$
\begin{aligned}
P_h(\textsc{Hits}(i,\ell)) &= P_h(X \geq \ell) \\
&\leq exp\left(-\frac{(1-\lambda)^2\ell^2}{2\lambda\ell + \frac{2(1-\lambda)\ell}{3}}\right) \\
&= exp\left(-\frac{(1-\lambda)^2\ell}{2\lambda + \frac{2(1-\lambda)}{3}}\right) \\
&= exp\left(-\frac{3(1-\lambda)^2}{2+4\lambda}\cdot\ell\right) \\
&= e^{-const\cdot\ell}
\end{aligned}
$$

Thus, in total we get a summation over ℓ of a polynomial term divided by an exponential term, which—of course—yields a constant even if the summation is over all natural numbers. Quantitatively, we see:

$$
\begin{aligned}
\mathbb{E}_h T(k) &\leq O(1) + \sum_{\ell=1}^{n} \ell^2 \cdot P_h(\textsc{Hits}(i,\ell)) \\
&\leq O(1) + \sum_{\ell=1}^{n} \frac{\ell^2}{e^{const\cdot\ell}} \\
&= O(1)
\end{aligned}
$$

That completes the proof for first insertions. Since subsequent insertions take exactly the same time as the first insertion, the argument is complete for all insertions. Since getting a key also costs at most a constant more than the length of the chunk that the key hashes into, the same argument suffices for gets also. The time bounds for initialization are evident. Thus, the proof is complete. □

8.10.1 Lower Bound

A natural question, is whether we can get a *provably efficient* concurrent re-sizing hash table by enhancing the Algorithm 26 to expand and contract. In the following lower bound argument, we prove that due to a phenomenon called "randomness leaking", it is difficult to obtain such a result.

Theorem 8.10.2 (Randomness Leak)**.** *Let H be a concurrent hash table designed for one or more processes, run with an adaptive scheduling adversary, and satisfying the following properties:*

1. *From its current state, H allows at least m inserts without changing the current hash function h.*

2. *$H.\textsc{Get}_\pi(k)$ reads the hash-value $h(k)$ at some point in its execution.*

3. *There is a monotonically increasing function $f : \mathbb{N} \to \mathbb{R}$, such that if $n \in \mathbb{N}$ distinct keys $k_1, \ldots, k_n$ with the same hash value $\iota = h(k_1) = h(k_2) = \cdots = h(k_n)$ are inserted into H, then a subsequent $H.\textsc{Get}_\pi(k)$ for any key $k \notin \{k_1, \ldots, k_n\}$ with $h(k) = \iota$ will take time at least $f(n)$.*

Then, the following statement holds:

- *For every $n \leq m$, and any length $\ell = \Omega(n \cdot m)$, assuming there are at least $mn + 1$ keys in the universe (i.e. key-space), there is a sequence of $O(\ell)$ Get and exactly n Insert operations that results in $\Omega(\ell \cdot f(n))$ work, i.e., $\Omega(f(n))$ work per operation on average.*

Proof. Fix the parameters m, n, and ℓ. The adversarial scheduler can produce the described work-heavy execution as follows. First, the scheduler chooses $mn + 1$ distinct keys $\kappa_1, \ldots, \kappa_{mn+1}$. In order to discover their hash values, the scheuduler asks process π to perform $\textsc{Get}_\pi(\kappa_i)$ for each $i \in [1, mn + 1]$. By the second assumption, the get-queries reveal all the hash-values $h(\kappa_i)$ to the adversarial scheduler. Since there are only m values in the range of the hash function, by the pigeonhole principle, some subset of $n + 1$ distinct keys must hash to the same value. Let $k_0, k_1, \ldots, k_n$ be such a subset of keys and let $\iota = h(k_0) = h(k_1) = \cdots = h(k_n)$ be their hash value. The adversarial scheduler now makes π perform $\textsc{Insert}_\pi(k_1, \perp), \ldots, \textsc{Insert}_\pi(k_n, \perp)$ in sequence. Finally, the adversarial scheduler makes π query $\textsc{Get}_\pi(k_0)$ repeatedly ℓ times.

The procedure above is sound, since by the first assumption the hash function h does not change as this procedure is executed. By construction, the total number of insertions is exactly n. The total number of Get operations is $mn + 1 + \ell = O(\ell)$, since $\ell = \Omega(mn)$. Finally, by the third assumption, each of the final ℓ query operations takes at least $f(n)$ time; thereby leading to a total work complexity of $\Omega(\ell \cdot f(n))$. □

Our lower bound shows that many concurrent hash tables require super-constant work per operation in the worst-case [182, 84, 60].[2]

[2]The paper on the "split-ordered hash table" [182] states that "under any scheduling adversary our new algorithm provides a lock-free extensible hash table with $O(1)$ average cost per operation" (page 383; re-expressed as Theorem 3.15 in that paper). However, from an inspection of the code, it is clear that the algorithm satisfies the hypotheses of Theorem 8.10.2 with $f(n) = n$, thus, in the worst-case operations require $\Omega(n)$ work. In particular, the original

8.11 Discussion and Future Work

In this work, we designed the first algorithms for concurrent fast arrays and fast generalized arrays. We have also presented some brief experiments to measure the empirical efficiency of our fast arrays.

Just as sequential fast arrays have found several applications, we envisage future work that explores applications of these concurrent fast arrays. The following directions seem promising.

- The concurrent union-find data structure of Jayanti and Tarjan [118], which is used in the fastest parallel algorithms for computing connected components and spanning forests on CPUs and GPUs [95, 51], requires a generalized array of n nodes, with each node initially pointing to itself, i.e., $f(i) = i$. So, any concurrent union-find object on which only $o(n)$ operations are performed benefits from the use of our fast generalized array.

- A concurrent (standard) fast array is useful for implementing an adjacency matrix, E, of a mutable sparse graph. In particular, adding or removing an edge (i, j) is implemented by writing 1 or 0 (respectively) in $E[i, j]$, and querying an edge (i, j) is a simple read of $E[i, j]$. The real saving lies in storing the graph initially. To store a sparse graph of $m \ll n^2$ edges, we initialize the matrix E with all-zero entries in just $O(1)$ time, and then add the m edges, one at a time. Thus, the entire graph is stored in just $O(m)$ time, instead of the usual $\Theta(n^2)$ time.

- Kanellakis and Shvartsman introduced the *write-all* problem, a version of which is stated as follows: given an array A of length m such that each entry $A[i]$ has an arbitrary initial value, devise an algorithm for p asynchronous processes to initialize each entry $A[i]$ to 0, such that no process returns before the initialization is complete. This problem has attracted a lot of research [125, 150, 31, 11, 84], especially since a write-all solution is a critical subroutine in some implementations of concurrent hash tables [68, 69, 182].

  Although the two problems are different, fast arrays and write-all share the quest to achieve "fast initialization". The difference is that write-all insists on physically initializing each array element, whereas a fast array promises only to create the illusion of initializing each element.

analysis does not account for randomness leaks, i.e., it assumes that a hash function mapping individual keys independently and uniformly when it first sees them implies that specific user-specified keys (which could be adversarially correlated through the hash function, if the program picks keys dynamically as in the lower bound) will be uniformly distributed. I believe the constant bound could hold for schedulers which cannot orchestrate randomness leaks.

Thus, initialization takes only $O(1)$ time with fast arrays, while it takes at least linear time in any solution of write-all. Consequently, if an algorithm that uses a write-all solution can instead be satisfied with a fast array, then the algorithm's speed can potentially improve.

- Allocating a hash table of size n requires $\Theta(n)$ time using conventional arrays (because of initialization). As a result, it has been difficult to implement efficient re-sizable lock-free hash tables [68, 69, 182]. Using fast arrays however, a new table of any size can be allocated in just $O(1)$ time. Exploiting this feature, we are in the process of designing a re-sizeable wait-free hash table that guarantees $O(1)$ average time for *find* and *insert* operations.

We look forward to the further development and deployment of these ideas by algorithmists and practioners alike.

Chapter 9

The Generalized Wake-up Lower Bounds

9.1 Introduction

Data structures are the fundamental building-blocks behind algorithms. In asynchronous shared memory computing, *linearizable wait-free* data structures based on registers and universal primitives (such as *compare-and-swap* and *LL/SC*) have become the gold standard, since they are essentially equivalent to atomic data structures [93, 89]. A good metric for data structure efficiency is *step complexity*, the total number of shared memory steps executed to complete a data structure operation. While researchers have developed very efficient sequential data structures for *stacks*, *queues*, *priority-queues*, *union-find* etc., designing similarly efficient shared memory data structures has remained an elusive task. Even the simplest data structures such as *counters* and *fetch-and-inc* suffer from logarithmic overheads in the number of processes, p.

Worst-case lower bounds have been shown for many data types (stacks, queues, fetch-and-add, etc.) [103]. However, analysis of sequential data types tells us that amortized complexity can often be very different from worst-case complexity: for example, the single operation worst-case cost of a union-find operation on an instance of size n is $\Theta(\log n / \log \log n)$ [25, 64], while its amortized cost is only $\Theta(\alpha(n, m/n))$ (inverse-Ackermann function) [198, 196, 64]. Yet amortized lower bounds for concurrent data types remain a largely under-studied area.

In this work we propose a new technique, which we term the *generalized wake-up technique*, for proving lower bounds on the *amortized* complexities of operations on linearizable data types

that are built from primitive variables supporting read, write, and compare-and-swap (CAS). Our technique is developed by generalizing Jayanti's *wake-up problem*, and extending his lower bound analysis to our generalized version of the problem [103]. Our main theorem deals with a situation in which p processes are executing in an asynchronous shared memory system, trying to collect information about how many other processes are "waking up" by taking at least one step. Given a decreasing sequence of numbers less than p $s = s_1, \ldots, s_p$, our main theorem shows that any algorithm for p asynchronous processes that uses variables supporting only read, write, and CAS, and guarantees that at least s_i processes figure out that at least i processes have taken one step for each $1 \leq i \leq p$ must have an asynchronous schedule that forces the total number of steps by all processes to be at least $\Omega\left(\sum_{i=1}^{p} \log s_i\right)$. Our lower bound holds even for randomized algorithms.

The main contribution of our work is the use of our generalized wake-up lower bound in several reductions showing lower bounds on the amortized step complexities of various data types. In particular, we show that the following amortized lower bounds hold against all deterministic and randomized algorithms for asynchronous shared memory systems with p processes that use variables supporting read, write, and compare-and-swap:

1. m operations on any union-find object with n items take $\Omega(m \log\log(np/m))$ steps in expectation.

   Remark: this is the first non-trivial lower bound for the concurrent Union-Find data type, and is the most involved of our reductions. We consider this the highlight of our work.

2. Define an r-relaxed counter as a fetch-and-increment object whose "fetch" returns a value that is within an additive r of the correct counter value. m operations on a $((1-\varepsilon)p/2)$-relaxed counter take $\Omega(m \log p)$ steps in the worst case.

   Remark: Our lower bound shows that Ellen and Woelfel's implementation of fetch-and-increment [57] and Jayanti's implementation of a counter [106] have optimal amortized step complexity. Furthermore, the $\Omega(\log p)$ overhead for counting in an asynchronous system is unassailable even for randomized algorithms, and even if the counter needs to be only approximately accurate.

3. Define an r-relaxed priority queue as an object that returns one of the top r items in priority order. For $r > 1$, this is a relaxation of the standard priority queue–which corresponds to the 1-relaxed priority queue. m *deleteMin* operations on a $((1-\varepsilon)p)$-relaxed priority queue take

$\Omega(m \log p)$ steps in the worst case. We also prove analagous results for the r-relaxed stack and r-relaxed queue objects.

The remainder of this chapter contains three sections. In Section 9.2 we discuss the specifics of our concurrency model, define the wake-up problem and our generalized version of it, and state our main theorem used to prove amortized lower bounds. In Section 9.3 we discuss the Union-Find data structure and prove an $\Omega(m \log\log(np/m))$ lower bound on the expected step complexity of any concurrent implementation of it. Finally, in Section 9.4 we present our lower bounds for fetch-and-increment objects, counters, approximate counters, stacks, queues, priority queues, and the relaxed versions thereof.

9.2 Concurrency Model and the Wake-Up Problem

We consider the shared memory asynchronous multiprocessor model of computing with p processes $P = \pi_1, \ldots, \pi_p$. We allow an infinite set of arbitrarily initialized shared variables V, each of which can have arbitrarily large size (even infinite size). In this model, each of the p processes has its own local memory and also has access to the collective shared memory. A *step* of the algorithm by a process π_i consists of an arbitrary sequence of primitive operations by π_i involving only local variables ended by a single primitive operation involving a shared memory variable. Exactly one process executes an operation in every discrete time step; the particular process π_i to execute a primitive operation at time step t is chosen by an adversarial scheduling function $\sigma : \mathbb{N} \to P$. σ gets to decide which process in P executes at time t by looking only at the past (including past randomness).

In our model we allow three atomic primitive operations on each variable $x \in X$: $read(x)$: returns the value of x, $write(x, v)$: updates x's value to v, $CAS(x, v, v')$: checks if $x = v$, if so it updates x's value to v' and returns true;otherwise it does not change x's value, and returns false. We allow additional strength in the model by allowing the *write* and *CAS* primitives to update x's value to a random value. That is, we allow v to be a random variable that is resolved to a realization at the execution time of the operation.

In this work, we will be interested in concurrent algorithms. A concurrent algorithm A assigns code to each process π_i. For a particular algorithm A and a schedule σ, we say that A runs under σ if processes execute the primitive shared memory steps of their algorithms in the order assigned by the scheduler σ. The *history* of a process π_i up to time t is the entire sequence of operations

performed by π_i and return values received by π_i up to time t. In particular, if the history of a particular process π_i is the same if t_1 steps of algorithm A are executed under schedule σ_1 or if t_2 steps of algorithm A are executed under schedule σ_2, then the two scenarios are indistinguishable to π_i.

9.2.1 Wake-Up

In his work [103], Jayanti proved worst and expected case step complexity lower bounds for various data types such as *queues*, *stacks*, or *fetch-and-adds*. His proof proceeds in two steps. First, he defines a simple problem called *wake-up*, and shows that any solution to this problem has a worst-case schedule that forces *some* process to take $\Omega(\log p)$ steps. Second, he reduces the wake-up problem to the various different data structure problems.

In the asynchronous scheduling model, no process knows when it and other processes will be given their first step. Jayanti's *wake-up* problem [103], requires one of the p processes to discover that all processes have been given at least one step. Formally, the problem is to give code for the p processes such that the following conditions are met:

1. (termination) Each process π_i returns either 0 or 1 in a bounded number of its own steps.
2. (truthfulness) If some process π_i returns a 1 at time t, then $\{\sigma(1), \ldots, \sigma(t)\} = P$.
3. (non-triviality) At least one process must return a 1.

Jayanti's work implies the following theorem

Theorem 9.2.1 (Theorem 6.1 in [103])**.** *Let A be a randomized algorithm that solves the wake-up problem for the processes in set P of size $p = |P|$. Then, there is a schedule σ, such that the processes of P executing A under schedule σ results in some process performing $\Omega(\log p)$ steps in expectation.*

Jayanti uses Theorem 9.2.1 to show some process must take $\Omega(\log p)$ steps in a worst-case schedule of any solution to wake-up, and uses reductions to prove that worst-case step complexities of various objects is $\Omega(\log p)$. If it were in fact the case that the total step complexity—total steps taken by all p processes—in a solution to wake-up is $\omega(p)$, then his reductions would imply non-trivial amortized lower bounds as well. Unfortunately, we show that this is not true by giving Algorithm 27, which solves wake-up in $O(p)$ total steps—a constant number per process—using the idea of a tournament.

We first present the algorithm when p is a power of 2; the generalization is immediate to those familiar with binary trees. Let B be a complete binary tree with p leaves, that initially has a value 0 in every node. Each process p_i has a *current node* x_i (initially the ith leaf, ℓ_i, of the tree). The algorithm is recursive, and at each current node π_i tries to CAS a 1 into its current node. If the CAS succeeds, π_i changes its current node to $x_i.parent$ if $x_i.sibling = 1$; otherwise, it terminates and returns 0. A process that successfully gets to the root of the tree returns a 1 to indicate the all processes are up.

Algorithm 27 The tournament tree algorithm to solve wake-up (code for process i).

```
1: procedure WakeUp(x = ℓ_i)
2:     if x is the root then return 1
3:     if Cas(x.value, 0, 1) then
4:         if x.sibling.value = 1 then return WakeUp(x.parent)
5:     return 0
```

Informally, the algorithm works since a node that makes its current node x can infer that every process with a leaf in x's subtree must have taken at least one step. Thus, a process that has the root as its current node can return that all processes have woken up. The $O(p)$ step complexity results since at most $O(1/2^h)$ fraction of the processes ever set their current node at height h in the tree. We state and prove the correctness and work complexity of Algorithm 27 below.

Lemma 9.2.2. *The tournament tree algorithm (Algorithm 27) solves wake-up in $O(p)$ total steps.*

Proof. We argue correctness of the algorithm by showing each of the three conditions separately, and then derive the step complexity.

1. (termination) Termination of the algorithm follows from the fact that the tree has bounded depth.

2. (truthfulness) Induction on the number of steps shows that the tournament maintains the invariant that if a process π_i's current node is x_i, then no descendant of x_i contains the value 0 in it. The base case is trivial since all processes start at the leaves, and the leaves have no descendants. The induction step follows since π_i makes $x_i.value = 1$ and checks that $x_i.sibling.value = 1$ before updating its current node to $x_i.parent$. In particular, the invariant implies that every π_j is awake for every leaf ℓ_j in the subtree of x_i, since π_j alone can set $\ell_j.value = 1$. Thus, when a process's current node is the root, it can safely return 1.

3. (non-triviality) The proof is trivial if $p = 1$. Otherwise, consider any internal node y in the tree. Induction on the height of the subtree at y shows that exactly one process successfully performs a CAS into the each of y's children nodes. We see that the latter of these processes to successfully perform its CAS on a child of y will surely make y its current node at some point in the algorithm. Since the root is an internal node, some process returns 1.

4. (efficiency) To show efficiency, we notice that half the processes that make it to height h in the tree fail to make it to height $h+1$. Since the work of a process is proportional to its final height in the tree, the total work is bounded up to a constant by $p + \sum_{h=1}^{\infty} ph/2^h = O(p)$.

□

Remark 9.2.3. If p is not a power of two, it suffices to use the almost complete binary tree of height $\lceil \log p \rceil$ and start some of the processes at leaves that are at depth $\lceil \log p \rceil - 1$.

9.2.2 Generalized Wake-Up

The $O(p)$ step complexity solution to wake-up in Algorithm 27 implies that the wake-up problem and reductions from it cannot be used to prove amortized lower bounds. But we dig deeper into the proof idea of Theorem 9.2.1 to reveal that a generalization of the wake-up problem can lead to a strengthening of Theorem 9.2.1 that is strong enough to show powerful amortized lower bounds. We give an informal description of this proof method below.

Consider the setting where the processes in the set P are running a randomized algorithm A which uses variables in the set V. At any point of time, we can define a *knowledge-set* for each process π_i. π_i's knowledge set at time t is the set of processes that π_i can infer to have woken up, i.e. taken at least one step, based on its history–its sequence of operations on shared memory and the values it received in return. Similarly, we can define a knowledge-set for each variable $x \in V$. x's knowledge set represents the set of processes that a process could infer to have woken up if it reads x. Since a process can return 1 only after it is sure that every other process has woken-up, the process that returns 1 must have a knowledge-set of size p before it returns. Jayanti proves his theorem by constructing a special schedule σ^* and a sequence of *"knowledge"* functions parametrized by round r: $K_r : (P \cup V) \to 2^P$, and shows that:

1. According to σ^*, each process that is yet to return takes exactly one step in each round r.

2. $|K_r(\Psi)|$ is an upper bound on the knowledge set of entity $\Psi \in (P \cup V)$ after round r.

3. $|K_r(\Psi)| \leq 4^r$ for any round r.

By the last inequality, Jayanti shows that when the algorithm A is run under schedule σ^*, there must be at least one process who does not return until round $r = \log_4 p$.

In order to get the correct generalization of the wake-up problem, we interpret the above schema in the following way.

Observation 1. The step complexity lower bound for wake-up comes about because a single process must reveal that it has a large knowledge set.

Therefore, in order to get a stronger lower bound, we define a generalization of wake-up which forces *multiple* processes to reveal that they have large knowledge sets. Thus motivated, we define the generalized wake-up problem below.

The *Generalized Wake-Up (GWU)* family of problems for p processes is parametrized by a monotonically non-increasing sequence $s_1, \ldots, s_p$ of values between 1 and p. The problem $GWU(s_1, \ldots, s_p)$ is to design an algorithm for the p processes such that each process π_i returns an integer in the range $[1, p]$ (a lower bound on the the number of processes it knows to have woken up), which satisfies the following conditions:

1. (termination) Each process π_i must return a value in the range $[1, p]$.
2. (truthfulness) If π_i returns k at time t, then $|\{\sigma(1), \ldots, \sigma(t)\}| \geq k$.
3. (non-triviality) At least k processes must return a value greater than s_k.

Informally, generalized wake-up specifies a sequence of values, and the algorithm must ensure that the processes demonstrate *at least* as much knowledge as the sequence requests.

Observe, that Jayanti's wake-up problem is the instance $GWU(p, 1, 1, \ldots, 1)$. Also, note that there is no algorithm with bounded step complexity for instances of generalized wake-up where $s_i > p - i + 1$, since the scheduler can wait for an unbounded amount of time before letting the $(p-i)th$ process wake up. So, we call the instance $GWU(p, p-1, p-2, \ldots, 1)$ the *Strong Wake-Up (SWU)* problem, and in general require that the sequence s be majorized by the sequence of descending numbers from p.

The following is our main theorem about Generalized Wake-Up.

Theorem 9.2.4. *Let A be a randomized algorithm that solves $GWU(s_1, \ldots, s_p)$ for processes $\pi_1, \ldots, \pi_p$. There is an asynchronous schedule σ such that when algorithm A is run under schedule σ, the total number of steps performed by all the processes is $\Omega\left(p + \sum_{i=1}^{p} (\log s_i)\right)$.*

Our proof of Theorem 9.2.4 is built off the work of Jayanti in [103]. We therefore summarize the important steps from Jayanti's work on the wake-up problem in the following discussion and then proceed to prove Theorem 9.2.4.

Consider the setting where the processes in the set P are running algorithm A which uses variables in the set V. Jayanti constructs a schedule σ^* and a sequence of *"knowledge"* functions parametrized by round r: $K_r : (P \cup V) \to 2^P$, and shows the following theorem.

Theorem 9.2.5 (Jayanti's Theorem: Lemma 5.1 and Lemma 5.2 from [103])**.** *Let the processes P be running some algorithm A:*

1. *σ^* operates in rounds where each process gets to take a step in every round, i.e., for each non-negative value $r \in \{0\} \cup \mathbb{N}$, $\{\sigma^*(rp+1), \ldots, \sigma^*(rp+p)\} = P$.*

2. *Knowledge is initially small and grows at most exponentially: $\forall \alpha \in V \cup P, |K_r(\alpha)| \leq 4^r$.*

3. *Knowledge is maximal (Indistinguishability): If the current time $t \in [rp+1, rp+p]$ (we are in the r^{th} round), then for each process π, there is an alternate schedule $\sigma_{\pi,t}$ and an alternate time t' such that $\{\sigma_{\pi,t}(1), \ldots, \sigma_{\pi,t}(t')\} = K_r(\pi)$, and running the same algorithm A with schedule $\sigma_{\pi,t}$ for t' time steps as opposed to with σ^* for t time steps would yield the same history for process π. So, π cannot distinguish whether all process have taken steps or only those in the set $K_r(\pi)$.*

An informal interpretation of *Jayanti's Theorem* is as follows: At any point of time, we can define a *knowledge-set* for each process π_i. π_i's knowledge set at time t is the set of processes that π_i can infer to have woken up, i.e., taken at least one step, based on its history–its sequence of operations on shared memory and the values it received in return. Similarly, we can define a knowledge-set for each variable $x \in V$. x's knowledge set represents the set of processes a process could infer to have woken up if it reads x. Theorem 9.2.5 upper bounds the sizes of these knowledge sets as follows. $K_r(\Psi)$ is entity Ψ's knowledge set after round r, where Ψ is either a process in P or a variable in V. At the beginning of the algorithm each process only knows that it has woken up, and nothing about the rest of the processes; similarly, the variables are in their initial states and therefore do not indicate any information about which processes have woken up and taken a step. So, $|K_1(\Psi)| \leq 1$ for all $\Psi \in (P \cup V)$. The crux of the theorem is the last part, which states that Jayanti's schedule, σ^*, ensures that the amount of knowledge of any given entity Ψ at most quadruples per round.

We use Jayanti's Theorem to prove the lower bound of Theorem 9.2.4 on the total number of steps to solve the $GWU(s_1, \ldots, s_p)$ problem.

Proof of Theorem 9.2.4. If a process π_i returns a value v after r rounds of σ^*, its return value can be at most $v \leq |K_r(\pi_i)| \leq 4^r$, since otherwise by the indistinguishability clause of Jayanti's Theorem, it would also return the same value v in $\sigma_{\pi_i,t}$ and violate the safety condition of generalized wake-up. Since the return values of the processes must majorize the sequence $s_1, \ldots, s_p$, the process with the j^{th} largest return value cannot return before $\log_4(s_j)$ rounds. So, the total number of steps by all the processes must be at least the sum $\sum_{i=1}^{p} \log_4(s_i) = \Omega(\sum_{i=1}^{p} \log s_i)$. Moreover, every process must take at least one step in order to satisfy termination, and thus the total step complexity is also at least $\Omega(p)$. Combining the two bounds proves the theorem. □

The next corollary shows that this theorem is sufficient to get a non-trivial total step complexity for Strong Wake-Up.

Corollary 9.2.6. *The total number of steps by all processes to solve Strong Wake-Up is* $\Omega(p \log p)$.

Proof. $\sum_{i=1}^{p} \log(p - i + 1) = \log p! = \Omega(p \log p)$. □

While it relies on a simple observation, Theorem 9.2.4 yields powerful amortized lower bound results. In the remainder of this chapter, we describe some important data types and show amortized lower bounds on their operation step complexities.

9.3 Union-Find Lower Bound

The Union-Find object maintains a set, S, of n elements as a collection of equivalence classes, under dynamic unions. Initially, each element of the set is in its own individual equivalence class, and the single element is the *representative* of its equivalence class. The object supports three operations, defined as follows (where $x, y \in S$):

(1) UNITE(x, y): combine the equivalence classes containing x and y into a single equivalence class, and fix some element of this class to be the new representative;

(2) FIND(x): return the representative of x's equivalence class;

(3) SAMESET(x, y): return true if x and y are currently in the same equivalence class, and false otherwise.

The exact formulations of the object semantics have varied. Usually, sequential data type literature has focused on objects requiring support for only the UNITE and FIND operations [67, 97, 198, 196, 64, 75], while concurrent data type literature has focused on UNITE and SAMESET [11, 118]. We will follow this convention without restatement as we discuss the literature. Notably, our lower bound holds even if the concurrent implementation only supports UNITE and SAMESET. We also note that the various formulations of the problem are closely related. For instance, SAMESET(x, y) can be implemented trivially in the sequential setting by simply checking if FIND$(x) =$ FIND(y). While it is more difficult in the concurrent setting, Anderson and Woll show a very efficient implementation of SAMESET from FIND in [11].

The Union-Find object spurred the genesis of many sophisticated algorithmic and complexity analysis techniques. In 1964, Galler and Fisher designed a *compression* heuristic to improve the performance of the simple tree algorithm for the Union-find problem [67]. This algorithm spurred a decades-long investigation into the amortized complexity of operations on the union-find data type which produced fascinating advances in algorithmic analysis and lower bound techniques [67, 97, 198, 196, 25, 64, 11, 75, 118]. Amortized analysis has focused on the time complexity of performing m operations on a problem instance with n nodes. In 1973 Hopcroft and Ullman showed that the compressed-trees algorithm has a complexity of $O(m \log^* n)$ [97], where $\log^*$ is the *iterated logarithm*. In a breakthrough result, Tarjan showed tight bounds on the complexity of the same algorithm; his bound—$\Theta(m\alpha(n, m/n))$—related the simple problem of equivalence class maintenance to the complicated inverse-Ackermann function "α" [198]. Tarjan's analysis however, left open the question of whether there was some other better algorithm for the union-find problem. In attempts to refute such a possibility, Tarjan showed that no algorithm in a large class of *separable pointer machine algorithms* could beat the $\Omega(\alpha(n, m/n))$ lower bound [197], and Tarjan and van Leeuwen also proved that many variants of the original compressed tree algorithm matched the lower bound [196]. In 1984, Blum showed that while Tarjan and van Leeuwen's algorithms were possibly optimal in their amortized complexity, the worst-case complexity of Union-Find algorithms could be improved from $O(\log n)$ to $O(\log n / \log \log n)$ [25]; he also gave a matching lower bound in a restricted model of computation. Time complexity questions on both the amortized and worst-case time complexities of the union-find object were resolved in 1989 by Fredman and Saks. They gave lower bounds of $\Omega(\alpha(n, m/n))$ and $\Omega(\log n / \log \log n)$ for the amortized and worst-case complexities of Union-Find operations in the *cell probe* model of computation (which is stronger than standard RAM model) [64].

Investigation into the complexity of the linearizable wait-free shared memory union-find object was started by Anderson and Woll. Complexity of the concurrent algorithms has been measured in *amortized work*, the total number of primitive steps required for m operations divided by the number of operations m. In 1991, Anderson and Woll gave an algorithm with an amortized work complexity of $\Theta(p + \alpha(n, m/n))$ [11]. In 2016, Jayanti and Tarjan showed a randomized algorithm (that under certain scheduling assumptions) reduced the overhead per operation in the number of processes from $\Omega(p)$ down to $O(\log p)$ in expectation [118]. In concurrent work, Jayanti and Tarjan have showed the same expected work upper bound without any scheduling assumptions [personal communication]. The amortized complexity of an operation in Jayanti and Tarjan's algorithm is $O(\log(np/m) + \alpha(n, m/(np)))$. Compared to the sequential algorithm, their algorithm has logarithmic overhead in the number of processes p, but before this work a lower bound on the amortized complexity of the concurrent union-find data type (beyond the sequential lower bound of Fredman and Saks) was unknown.

In the subsequent subsection, we give the first non-trivial lower bound on the amortized work complexity of the concurrent union-find data type. In particular, we show the existence of a set of m union-find operations performed across p processes and a schedule σ such that any linearizable algorithm for concurrent union-find must perform at least $\Omega(m \log \log(np/m))$ steps to perform the operations.

9.3.1 Amortized Lower Bound

In order to prove a lower bound, we must find a way of leveraging the generalized wake-up lower bound. That is, processes must leverage the UNITE and SAMESET operations to get information about how many other processes have woken up and taken at least one step. Our high-level idea for gaining this information is to leverage path lengths in graphs. That is, let V be a set of vertices, and $E \subseteq V \times V$ be a set of edges (generally not including all possible edges). A process p can dynamically "add the edge $(u, v) \in E$" into the graph by performing UNITE(u, v). After some processes have added some edges, there is a resultant graph $H = (V, F)$, where $F \subseteq E$ is the set of edges that have already been added to the graph. A process p' can "check if vertices u' and v' are connected" in the resultant graph H by performing SAMESET(u', v'). We observe that if SAMESET(u', v') is true, then at least $\delta(u', v')$ edges must have been added to the graph, where δ is the shortest paths function on the original graph $G = (V, E)$. In particular, if each process is somehow restricted to add at most one edge, then SAMESET$(u', v') = true$ implies that at least

$\delta(u', v')$ have woken up. On the flip side, also note that if SameSet(u', v') is false, then that gives us no wake-up information. So, in order to get an amortized bound, we must design a procedure to ensure that many processes will get non-trivial wake-up information, so that we can leverage the generalized wake-up lower bound.

We will see as the proof unfolds that ensuring many SameSet operations return true amounts to picking a highly connected graph G. However, the length of the shortest paths become smaller as connectivity goes up. For instance, in the best connected graph–the complete graph–all shortest paths are of length one, and thus even when SameSet operations return true, only trivial wake-up information is gained. This inherent trade-off between connectivity and long paths makes the graph we pick very pivotal to the proof. We formalize the notion of graph connectivity that we will require:

Definition 9.3.1. The edge expansion constant $h(G)$ of a graph $G = (V, E)$ is defined as

$$h(G) \triangleq \min_{\substack{S \subset V \\ |S| \leq |V|/2}} \frac{|E_{out}(S)|}{|S|},$$

where $E_{out}(S)$ is the set of edges that have one vertex in S, and one vertex in $V - S$.

The edge expansion constant measures the connectivity of a graph – the higher the constant, the more connected the graph is. Sparse graphs with a high edge expansion constant are called *expanders*, and are known to have many applications in computer science [96]. The following result implied by Friedman [66] states that good expanders exist for any number of vertices:

Lemma 9.3.2 (By Friedman [66])**.** *For every large-enough n, there is a 4-regular expander graph $\mathcal{G}_n = (V, E)$ with $|V| = n$, $|E| = 2n$, and $h(\mathcal{G}_n) > 0.2$.*

In the following lemma, we prove that all subgraphs of an expander with sufficiently many edges have large connected components.

Lemma 9.3.3. *There are constants $a < 1$ and $b > 0$ such that the following holds for any large enough n. Let $H = (V, F)$ be a subgraph of $\mathcal{G}_n = (V, E)$. If $|F| \geq a|E|$, then the largest connected component of H has size at least $b|V|$.*

Proof. Let $C_1, \ldots, C_\ell \subset V$ be the connected components of H. Assume none of the components are larger than $|V|/2$ in size. Then, by the definition of $h(\cdot)$, at least $h(\mathcal{G}_n) \cdot |C_i|$ edges of E go

between C_i and $V - C_i$. Thus,

$$|E| - |F| \geq \frac{1}{2}\sum_{i=1}^{\ell} |E_{out}(C_i)| \geq \frac{1}{2}\sum_{i=1}^{\ell} h(G)|C_i| = \frac{h(G)|V|}{2} = \frac{h(G)|E|}{4}.$$

So $|F| \leq |E|(1 - h(G)/4) < 0.95|E|$ if $\max_i |C_i| \leq 0.5|V|$.

Set $a = 0.95$ and $b = 0.5$, and the result follows. □

We will now construct a reduction from Union Find to Generalized Wake-Up. For any integer $t > 0$ and large enough n so that $\mathcal{G}_n$ has the properties of Lemma 9.3.2 and Lemma 9.3.3, let $p = 2n^3t$ and $m = 2p$. We will now describe an algorithm on p processes that performs m operations of concurrent disjoint set union on n nodes.

Work with $\mathcal{G}_n = (V, E)$. Define a *process type* to be a triple $(e = (u, v), x, y)$ where $e \in E$ and $u, v, x, y \in V$. Since $|E| = 2|V| = 2n$, there are a total of $2n^3 = p/t$ process types, so we can create t processes of each type. The type $((u, v), x, y)$ process performs the steps in Algorithm 28.

Algorithm 28 A reduction to generalized wake-up

```
1: procedure GWU(((u, v), x, y))
2:     UNITE(u, v)
3:     return max(δ(x, y), 1) × SAMESET(x, y)
```

Order the processes as $\pi_1, \ldots, \pi_p$ based on the linearization points of completing the UNITE operation on line 2. We denote the type of π_i by $T(\pi_i) = (e_i = (u_i, v_i), x_i, y_i)$. Let $I = [1, \ell]$, be the indices of the first ℓ processes to linearize their UNITE operations; and define $H = (V, F)$ where $F = \bigcup_{i \in I}\{e_i\}$ is the set of edges. The constant ℓ will be defined by the schedule to be the smallest constant that ensures a "large" connect component in H by Lemma 9.3.3. In particular, pick ℓ to be the minimum number such that $|F| \geq a|E|$. Let C be the set of vertices in the largest connected component of H. Let $P \triangleq \{\pi_i : e_i \notin F\}$ be the set of processes whose UNITE operation in Algorithm 28 will not attempt to add an edge in F. Finally, let a, b be the constants from Lemma 9.3.3.

We will now prove a series of short lemmas that have the following informal meanings:

1. The largest connected component in $H = (V, F)$, i.e. C, contains a constant fraction of the vertices of V. Furthermore, the set of processes in P which will attempt to unite edges not in F, is at least a constant fraction of the total number of processes.

2. If x and y are random vertices in V, then with constant probability they will be in the connected component C and the distance between them will be $\Omega(\log n)$.

3. The first two lemmas imply that a constant fraction of the processes will return a value that is $\Omega(\log n)$.

4. The previous lemma implies an $\Omega(\log \log n)$ amortized lower bound.

Lemma 9.3.4. *$|C| \geq bn = \Omega(n)$ and $|P| \geq (1-a)p - p/|E| = \Omega(p)$.*

Proof. $|C| \geq b|V| = bn$ by Lemma 9.3.3, since $|F| \geq a|E|$.

Since there are $p/|E|$ processes that are trying to perform a union involving the vertices of any given edge in G, $|P| \geq (|E| - \lceil a|E| \rceil) \cdot \frac{p}{|E|} \geq p(1-a) - p/|E|$. □

Lemma 9.3.5. *Let (x, y) be a random pair of vertices from V.*

$$\mathbb{P}(x, y \in C \text{ and } \delta(x, y) > (1/2)\log_4 n) \geq b^2(1 - 2/(b\sqrt{n})).$$

Proof. $\mathbb{P}(x, y \in C) \geq b^2$ because $|C| \geq b|V|$ by Lemma 9.3.4. And $\mathbb{P}(\delta(x, y) > (1/2)\log_4 n \mid x, y \in C) > 1 - 2/(b\sqrt{n})$, since by the 4-regularity of $\mathcal{G}_n$, the radius $r > 1$ neighborhood of vertex x has size at most $2 \cdot 4^r$. So, the $r = (1/2)\log_4 n$ neighborhood of x has at most $2\sqrt{n}$ vertices. □

Lemma 9.3.6. *A constant fraction of the processes return a value that is $\Omega(\log n)$.*

Proof. All the processes $\pi_i \in P$ for which $x_i, y_i \in C$ and $\delta(x_i, y_i) \geq (1/2)\log_d n$, return at least $(1/2)\log_d n = \Omega(\log n) = \Omega(\log p)$, since $i > \ell$ by the definition of P. By Lemma 9.3.5, the fraction of such processes returning $\Omega(\log n)$ is at least $b^2(1 - 2/b\sqrt{n})$. So the total number of such processes is at least $|P|b^2(1 - 2/b\sqrt{n}) \geq \Omega(p)$, by Lemma 9.3.4. □

Lemma 9.3.7. *Algorithm 28 solves $GWU(s_1, \ldots, s_p)$ where $s_1 = \cdots = s_\kappa = \Omega(\log n)$, $s_{\kappa+1} = \cdots = s_p = 1$ and $\kappa = \Omega(p)$.*

Proof. We argue each of the *GWU* conditions. Termination follows by wait-freedom of the Union Find data structure. Truthfulness follows since every process adds at most a single edge to the graph through its unite operation on line 2, so a successful SameSet(x, y) operation indicates that at least $\min(1, \delta(x, y))$ processes have woken up. And non-triviality follows by Lemma 9.3.6. □

We now translate the lower bound from Lemma 9.3.7 into a lower bound parametrized by the number of operations m, the number of processes p, and the size of the Union-Find object n, through a combinatorial argument.

Theorem 9.3.8. *Consider any randomized algorithm for p-process shared memory Union Find on an instance with n nodes. Let $m > n$ and $m > 2p$. Then, there is a sequence of m Union Find operations on the p processes and n nodes that takes $\Omega(m \log\log(np/m))$ shared memory steps.*

Proof. Divide the n nodes into groups of size $s = \lfloor 2np/m \rfloor$. Have $g = \lfloor m/2p \rfloor$ groups $V_1, \ldots, V_g$, ignoring extra nodes.

For each group V_i, do not use all the nodes in the group, but only use $s' = \Theta(s^{1/3}) = O(p^{1/3})$ of the nodes, so that we can choose t so that $p' \triangleq 2(s')^3 t = \Omega(p)$ and $p' \leq p$, and can run Algorithm 28 on p' processes, where there are t processes of each type. By Lemma 9.3.7 and the GWU lower bound (Theorem 9.2.4), running Algorithm 28 with these parameters takes $\Omega(p' \log\log s') = \Omega(p \log\log(np/m))$ shared memory operations in the worst case.

We therefore run Algorithm 5 as above for each group of nodes, in a series of g rounds, letting all operations from the previous round complete before starting the next one. The total worst-case cost is $\Omega(gp \log\log(np/m)) = \Omega(m \log\log(np/m))$, the total number of processes used is $\leq p$, the total number of nodes used is $\leq n$, and the total number of Union Find operations invoked is at most $2p' \cdot g \leq 2p \cdot m/(2p) = m$. □

9.4 Other Lower Bounds

The complexity of a linearizable concurrent data structure is, in general, bounded below by the complexity of its sequential counterpart since the concurrent data structure must deal with coordinating between the multiple processes. Lots of recent algorithmic work has focused on designing concurrent data structures with low concurrency overheads. Two notable examples of concurrent data structures that have been incredibly successful in reducing concurrency overhead to a term that is merely logarithmic in the number of processes are: Ellen and Woelfel's algorithm for implementing the *fetch-and-increment* object [57], and Jayanti's algorithm for implementing concurrent *counters* [106]. Both of these works have produced concurrent data structures with worst-case and amortized step complexities of merely $O(\log p)$. Jayanti's lower bound in [103] is sufficient to show that these implementations have optimal worst-case complexities. In this section, for the first time, we show that the amortized time complexities of these implementations are also optimal.

Where research has failed to produce concurrent data structures with low concurrency overheads, researchers have attempted to produce fast data structures for relaxed versions of the same data structures. For example, an h-relaxed Priority Queue is a version of a priority queue where the *deleteMin* operation is required to return an element in the smallest h elements in the priority queue, rather than just the smallest element. A notable recent work in this direction is the SprayList, a probabilistic implementation of a $O(p \log^3 p)$-relaxed priority queue, which supports *deleteMin* in $O(\log^3 p)$ expected steps [4, 5]. In this section, we show amortized step complexity lower bounds for relaxed priority queues as well as for queues and stacks.

We start by giving descriptions of the data types of interest.

1. **Counters:** objects that enable incrementing (or decrementing) and reading the current value.

   (a) *fetch-and-inc:* holds a single value v (initially 0) and supports the single operation F&Inc, which simultaneously returns the current value of v of the object and updates $v \leftarrow v + 1$.

   (b) *counter:* holds a single value v (initially 0), and supports two operations Inc(), which updates $v \leftarrow v + 1$, and Read, which returns the current value v.

   (c) *h-approx-counter:* A counter whose Read operation is allowed to return a value in the range $[v - h, v + h]$.

2. **Ordered Collections:** objects that allow insertion and deletion according to a standard order. Examples include Queues, Stacks, and Priority queues. We always let Insert be the insert operation (enqueue, push, insert) and Remove the removal operation (dequeue, pop, deleteMin).

   An h-relaxed Ordered Collection is an ordered collection object whose Remove operation is allowed to return one of the top h items in the standard order (as opposed to just the top one).

We give amortized lower bounds for counters and ordered collections in this section. When appropriate, we also remark on algorithms with matching or near-matching step complexities.

Theorem 9.4.1 (Counters)**.**

(1) Let A be any linearizable randomized algorithm for fetch-and-inc. For every $m \geq p$, there is a sequence of m operations and a schedule that forces A to take $\Omega(m \log p)$ steps to perform the operations.

(2) Let A be any linearizable randomized algorithm for a $p(1-\varepsilon)/2$-approx-counter, where $\varepsilon > 0$ is a constant. For every $m \geq 2p$, there is a sequence of m operations and a schedule that forces A to take $\Omega(m \log p)$ steps to perform the operations.

Proof. To prove (1) in the case that $m = p$, we observe that the p processes can solve strong wake-up using fetch-and-increment, by each returning x.F&Inc on a fetch-and-increment object initialized to 0. If $m > p$, we repeat the procedure in phases. That is, the p processes use p operations to solve SWU in each phase, and there is a break in between phases to ensure that all processes complete their operations from the previous phase. Simply running for m/p phases yields the lower bound when $m > p$.

We prove (2) in the specific case of the $h = p/3$ approx-counter for concreteness. The proofs for all ε are exactly analagous. First, let $m = 2p$. We consider the algorithm where each process π_i does: x.Inc followed by x.Read, on an h-approx-counter object x. If a process reads a value of at least $p/2$, it knows for sure that at least $p/2 - p/3 = p/6$ processes have woken up, and thus can return $p/6$. Every process after the $p/2 + p/3 = 5p/6$th process to perform increment, must read a value of at least $p/2$ by linearizability, and thus at least $p/6$ processes return a value of at least $p/6$ thereby solving $GWU(s)$, where $s_i = p/6$ if $i \leq p/6$ and $s_i = 1$ otherwise. Theorem 9.2.4 gives an $\Omega(p \log p)$ step complexity lower bound for this GWU instance. If $m > 2p$, simply run in phases. □

Remark 9.4.2. Theorem 9.4.1 shows the tightness of two algorithms: (1) Ellen and Woelfel give a $O(m \log p)$ linearizable wait-free *fetch-and-increment* construction in [57], and (2) Jayanti gives a $O(m \log p)$ construction for a counter in [106]. (A counter is obtained by f-arrays when f is the operation "+".)

The argument in Theorem 9.4.1 also shows that maintaining the size of a shared memory data structure (such as a stack, queue, or set), even up to an additive approximation, is impossible without at least an amortized-logarithmic step complexity in either the Insert or Remove method. The following theorem shows that if the collection is ordered, then just the Remove method must have a logarithmically high amortized complexity even if size does not need to be maintained.

Theorem 9.4.3. *Let A be any linearizable randomized algorithm for an h-relaxed Queue, h-relaxed Stack, or h-relaxed Priority Queue for $h = (1-\varepsilon)p$, where $\varepsilon > 0$ is a constant. For every $m \geq p$, there is a set of m Remove operations and a schedule that forces A to take $\Omega(m \log p)$ steps to perform the operations.*

Proof. For simplicity of exposition, we start with the case of $\varepsilon = 1/2$ and $m = p$, so that $h = p/2$ and consider the h-relaxed priority queue. Let Q be the priority queue, and let it initially be filled with the items $\{1, \ldots, m = p\}$, and consider the algorithm in which each process π_i simply performs Q.Remove. If a process receives a value greater than $3p/4$, then it must be the case that at least $p/4$ processes have already succeeded in performing their Remove operation—otherwise, $3p/4$ would not be among the $p/2$ smallest items. Since there are only p items in Q and there are p removals, every item is removed by some process. It follows that the p processes can solve $GWU(s)$ with $s_i = p/4$ if $i \leq p/4$ and $s_i = 1$ otherwise. This yields a lower bound of $\Omega(p \log p)$ by Theorem 9.2.4.

If $m > p$, then simply start with a priority queue with the items $\{1, \ldots, m\}$, and run in m/p phases as in the proof of Theorem 9.4.1. Note that in this case it is possible that the item "1" will remain in Q until the final phase, since it may never be picked by a removal due to the relaxation, and thus it cannot be guaranteed that the p items that are removed by the processes in phase r are $(r-1)p + 1, \ldots, rp$. However, the processes can communicate "out of band" between phases, and thus know exactly which items are remaining in Q at the beginning of each phase.

If ε is not $1/2$ but is some other constant, the proof proceeds similarly. The main difference is that the instance of $GWU(s)$ that is solvable now has $s_i = \varepsilon p/2$ for $i \leq \varepsilon p/2$ and $s_i = 1$ for $i > \varepsilon p/2$.

If the object Q is not a priority queue, but is instead a stack or a queue, then we simply start with the items $\{1, \ldots, m\}$ in last-in-first-out (LIFO) / first-in-first-out (FIFO) order. □

అధ్యాయము 10

సామాన్య జాగృతిపరిష్కారం

జాగరితస్థానో బహిష్ప్రజ్ఞః సప్తాఙ్గ ఏకోనవింశతిముఖః స్థూలభుగ్వైశ్వానరః ప్రథమః పాదః[201]

సఙ్క్షేపః ఎతసమిన్ లెఖె సమకాలిక వినాప్రతీక్ష రెఖీయ దత్తాంశసంవిధానానామ్ సంశ్రమాధొబన్ధాః పరీక్షామహె। వాఙ్మయె సమకాలిక దత్తాంశసంవిధానానామ్ ప్రతికూలతమ శ్రమవిషమతాధొబన్ధాః స్థాపనార్థమ్ ప్రసాద్ జయన్తివర్యస్య జాగృతిపరిష్కారమ్ ఎకమ్ ముక్యపరికరమ్। కిన్తు, తామ్ జాగృతిపరిష్కారసమస్యామ్ n సంసాధకాని $O(n)$ సంశ్రమెణ పూరయితరమ్ విధికల్పమ్ ప్రదృశ్య, తతః తెన పరికరెణ సంశ్రమాధొబన్ధనిరూపణమ్ దుర్లభమ్ ఇతి ప్రదర్శయామహె। సంశ్రమాధొబన్ధనిరూపణార్థమ్ సామాన్యజాగృతిపరిష్కారమ్ ఇతి నవ పారమితిక సమస్యశ్రెణిమ్ పరిభాష్య, తమ్ పరిశీల్య, శ్రెణ్యౌ ప్రతిసమస్యాయామ్ సంశ్రమాధొబన్ధమ్ స్థాపయామః। శ్రెణౌ కష్టతమసమస్యామ్ పూరణార్థమ్ n-సంసాధకసంవిధానస్య $\Omega(n \log n)$ సంశ్రమ ఆవశ్యకమ్। ఎతత్ ప్రమెయమ్ ఉపయుజ్య, న్యూనీకరణద్వార వివిధ బహుసంసాధకవస్తూపవిధెః సంశ్రమాధొబన్ధాః నిరూపయామః। తదనన్తరమ్, శ్రెణౌ కానిచన ముఖ్యసమస్యానామ్ అధొబన్ధాః ఎవ ఉద్బన్ధాః ఇతి నిరూపయామః।

సారాంశం ఈ లేఖలో సమకాలిక వినా-ప్రతీక్ష రేఖీయ దత్తాంశసంవిధానాల (కృత్రిమాణు వస్తువుల) సంశ్రమ అధోబంధాలను పరికిస్తాం. వాఙ్మయములో, సమకాలిక ప్రతికూలతమ శ్రమవిషమత అధోబంధాలను స్థాపించేడానికి జయంతి జాగృతి పరిష్కారం ఒక ముఖ్య పరికరం. కానీ జయంతి జాగృతి-పరిష్కార సమస్యను n సంసాధకాలు $O(n)$ సంశ్రమతో పూరించే విధికల్పమును ప్రదర్శించి, తద్వారా ఆ పరికరంతో సంశ్రమ అధోబంధ నిరూపణ దుర్లభం అని ప్రదర్శిస్తాం. సంశ్రమ అధోబంధాలను నిరూపించేడానికి తోడ్పడే సామాన్య-జాగృతి-పరిష్కారం అనే క్రొత్త పారమితిక సమస్య స్రేణిని నిర్వచించి, దాన్ని పరిశీలించి, శ్రేణిలోని ప్రతీ సమస్యకు సంశ్రమ అధోబంధాన్ని నిరూపిస్తాం. శ్రేణిలో కష్టతమమైన సమస్యను n సంసాధకాల సంవిధానము పూరించాలంటే $\Omega(n \log n)$ సంశ్రమ ఆవశ్యకం. ఈ సిద్ధాంతాన్ని ఉపయోగించి, న్యూనీకరణలద్వారా వివిధ బహుసంసాధక వస్తు ఉపవిధులకు సంశ్రమ అధోబంధాలను నిరూపిస్తాం. ఆ పై, శ్రేణిలో కొన్ని ముఖ్యసమస్యలకు మా అధోబంధాలే ఉద్బంధాలని రచనాత్మకంగా నిరూపించి, తద్వారా మా ఆ అధోబంధాలను బలపరచేట అసాధ్యమని గమనిస్తాం.

10.1 ఉపోద్ఘాతము

వేగమైన విధికల్పాలకు వెన్నెముకలు దత్తాంశసంవిధానాలు. సమకాలికసంగణనములో CASతో కల్పించేన రేఖీయ [93] వినా-ప్రతీక్ష [89] దత్తాంశాలు సువర్ణప్రమాణాలు. అందున, ఇలాంటి **కృత్రిమాణు**దత్తాంశాల శ్రమవిషమత-ను నిర్ధారించేట ఒక ముఖ్యాంశం. శ్రమవిషమత ఉద్బంధాలను విధికల్ప పరిశీలనతో గ్రహించేచ్చు. శ్రమవిషమత అధోబంధాలను నిర్ధారించేటకు ఉపకరించే ఒక ప్రధాన ఆధునిక సాధనం జాగృతి-పరిష్కార సమస్య.

జాగృతి-పరిష్కార సంసాధక సమన్వయ కర్త్వమును జయంతి 1998లో ప్రవేశపెట్టారు [103]. జాగృతి-పరిష్కారం ఒక n సంసాధక అసమకాలిక సంవిధానంలో నిర్వచింపబడినది. దీనిలో కేంద్రబిందువు, ఉన్న n సంసాధకాలలో ఒకటి మిగతా అన్ని సంసాధకాలు **లేచి** కనీసం ఒక అడుగు వేసాయని గ్రహించేలి.

ఈ జాగృతి-పరిష్కార సమస్యను కేవలం **పఠ** (read), **లిఖ** (write), **పోల్చివ్రాత**లతో (CAS) మాత్రం పూరించే ప్రతి విధికల్పములో ఏదో ఒక సంసాధకం ప్రతికూలతమ చలనలో $\Omega(\log n)$ శ్రమిస్తుందని జయంతి నిరూపించేరు. ఈ సిద్ధాంతాన్ని బట్టి న్యూనీకరణల ద్వారా వివిధ సమకాలిక-దత్తాంశ-సంవిధాన విధికల్పాల ప్రతికూలతమ ఉపవిధులకు అధోబంధాలు నిరూపించేగలిగారు.

ఈ లేఖలో, మేము సమకాలికదత్తాంశసంవిధానాల ప్రతికూలతమ సంశ్రమవిషమతను విచారిస్తాం. ఈ విచారణలో ప్రథమాంశముగా, జాగృతి-పరిష్కారపు సంశ్రమను విశ్లేషించి, ఈ సమస్యను కేవలం $O(n)$ సంశ్రమతో పూరించచ్చని విధికల్పనద్వారా నిరూపిస్తాం. పరిణామతః, జాగృతి-పరిష్కార న్యూనీకరణలద్వారా దత్తాంశసంవిధానాల సంశ్రమవిషమతలకు అధోబంధాలు స్థాపించేట దుర్లభం. ఈ తడను అధిగమించేటకు, జాగృతి-పరిష్కారమును సామాన్యపరచి, సామాన్య-జాగృతి-పరిష్కారం అనే క్రొత్త సమస్య శ్రేణిని ప్రవేశపెడతాం. ఈ శ్రేణిలో జయంతి నిర్వక్త జాగృతి-పరిష్కారం సులభతరమైన సమస్య, అందుచేత దన్ని **సులభ జాగృతి-పరిష్కారమని** పునర్నామకరణం చేస్తాం. జయంతి సులభ సమస్య విశ్లేషణను విస్తరించి, శ్రేణిలోని ప్రతి సమస్యకు అధోబంధాన్ని స్థాపిస్తాం. ఉదాహరణకు, శ్రేణిలోని కష్టతమమైన **కఠిన జాగృతి-పరిష్కార** సమస్యకు సంశ్రమ అధోబంధం $\Omega(n \log n)$. అధోబంధ స్థాపనానంతరం, న్యూనీకరణల ద్వారా వివిధ దత్తాంశసంవిధానాలకు సంశ్రమ అధోబంధాలను నిరూపిస్తాం. లేఖ తుది అంశముగా, అధోబంధాల దృఢీకరణశక్యతను విశ్లేషిస్తాం. ఈ దిశలో, ముఖ్యముగా కఠిన జాగృతి-పరిష్కారణ సంశ్రమ అధోబంధానికి సమమైన ఉద్బంధాన్ని విధికల్ప ప్రమాణముగా నిరూపిస్తాం.

10.2 యంత్రప్రతికృతి మరియు పూర్వాంశాలు

ఈ లేఖలో సంవిభక్త-స్మృతి గల అసమకాలిక బహుసంసాధకమును ప్రస్తుతిస్తాం. అట్టి సంగణక-సంవిధంలో $\mathbf{s}_1, \ldots, \mathbf{s}_n$ అనే n సంసాధకాలు, అసమకాలికముగా (అనగా వేరువేరు వేగాలతో అడుగులేసుకుంటూ)విధికల్పమును నిర్వర్తిస్తాయి. ప్రతి సంసాధకం $\mathbf{s}_i$ మొద్దట్టి అడుగు వేసినప్పుడు అది **లేచిందని** (అనగా **జాగృతం** అని) గణిస్తాము. లేచిన తరువాత, ప్రతి సంసాధకము విధికల్పములో దానికివ్వబడిన ఉపవిధిని కాలబంధము అవ-

కాశమిచ్చినప్పుడల్ల ఒక అడుగు వేస్తూ తిరుగిచ్చేదాక నిర్వర్తిస్తుంది. సంవిధము అసమకాలికము కనక, **రిపు-కాలబంధం** ఏ సంసాధకము ఎప్పుడు మరుసటి అడుగు వెయ్యాలో నిర్దేశిస్తుంది. అనగా, కాలబంధం $\mathbb{N}$లోని ప్రతి **కాలమాత్ర**కి, ఆ మాత్రలో ఏ సంసాధకము అడుగువేస్తుందో నిశ్చయిస్తుంది. కాలబంధం రిపువు కనక వివిధ సంసాధకాల అడుగుల మిశ్రమ-క్రమము ఏ విధముగానైనా ఉండచ్చు. ప్రతీ మాత్రలో సరిగ్గా ఒక సంసాధకము మాత్రం అడుగు వేస్తుంది కనక సంవిధములో ప్రతి సంసాధక అడుగుకు ఒక **సంవిధకాలమాత్ర** అని లెక్క. అలాగే, ఆ అడుగు వేసిన సంసాధకము **స**$_i$కి ఒక **శ్రమ** పడినట్టు లెక్క. సంసాధకాల శ్రమలను కూడితే ఒచ్చే సంఖ్యని సంవిధముయొక్క **సంశ్రమ** అని ప్రస్తుతిస్తాము. బహుసంసాధక సంగణక పరిభాషలో, ఒక ప్రత్యేక కాలబంధానుసారం సంసాధకాలచే నిర్వహింపబడ్డ విధికల్పమును **చలన** అంటారు.

ఈ భాషలో సుతథ్యముగా ప్రసాద్ జయంతియొక్క జాగృతి-పరిష్కారాన్ని ఇలా నిర్వచించచ్చు.

నిర్వచనం 10.2.1 (జాగృతి-పరిష్కారం)**.** ఒక n సంసాధక సంవిధానంలో **జాగృతి-పరిష్కారం** చేసే విధికల్పమునకు మూడు గుణములుండాలి:

1. **సమాప్తి:** ప్రతి సంసాధకము ఒక సదసత్తు (అనగా సత్, లేదా అసత్ ను) తిరుగివ్వాలి.
2. **సత్యవాక్కు:** అన్ని సంసాధకాలు లేచేదాకా (ఒక అడుగు వేసేదాక), ఏ సంసాధకము సత్తును తిరుగివ్వకూడదు.
3. **అవితండం:** అన్ని సంసాధకాలు అసత్తును తిరుగివ్వకూడదు.

ఆయన ఈ సమస్యను గురించి నిరూపించిన రెండు అధీబంధాలను ఇక్కడ పునఃస్మరిద్దాం:

సిద్ధాంతం 10.2.2 (అవిచక్షణీయత [103]లో Lemma 5.2)**.** *P అనే గణం లోని సంసాధకాలకు జాగృతి-పరిష్కారాన్ని* **వ** *అనే యాదృచ్ఛికవిధికల్పం కేవలం పఠ, లిఖ, పోల్చివ్రాతలు భరించే V అనే గణం వికారులతో పూరిస్తుందని భావిద్దాం. అప్పుడు క్రింది గుణాలను పూర్తి (అనగా 1) సంబవతతోగల కాలబంధం* **క** *ఉండితీరాలి:*

1. **క** *అనే కాలబంధం కల్పాల్లో చలిస్తుంది. ప్రతి కల్పంలో తిరుగివ్వని ప్రతి సంసాధకము సరిగ్గా ఒక అడుగు వేస్తుంది.*
2. *అ $\in P \cup V$ అనే ప్రతి వస్తువుకు, ప్రతి కల్పం k లో జ్ఞా$_k$(అ) $\subseteq P$ అనే సంసాధక ఉపగణం జోడింపబడి ఉంటుంది. ఈ గణాలను* **జ్ఞాన గణాలని** *సంబోధిస్తాం.*
3. **స** *అనే సంసాధకాన్ని ధ్రువపరుద్దాం.* **క** *కాలబంధ క్రమములో అన్ని సంసాధకాలు చలించినా,* **క** *కాలబంధ క్రమములో జ్ఞా$_k$(**స**) లోని సంసాధకాలు మాత్రము చలించినా* **స** *దృష్టిలో కనీసం $k^{\text{వ}}$ కల్పం వరకు అవిచక్షణీయం.*
4. $|\text{జ్ఞా}_k(\textbf{స})| \leq 4^k$

పై సిద్ధాంతాన్ని అనుసరించి వచ్చే అధోబంధము:

సిద్ధాంతం 10.2.3 (జయంతి ప్రతికూలతమ అధోబంధ సిద్ధాంతం [103]లో Theorem 6.1)**.** *P అనే గణం లోని సంసాధకాలకు జాగృతి-పరిష్కారాన్ని* **వ** *అనే యాదృచ్ఛికవిధికల్పం కేవలం పఠ, లిఖ, పోల్చివ్రాతలతో పూరిస్తుందని భావిద్దాం. $|P| = n$ ఐన, అప్పుడు, ఏదో ఒక కాలబంధం* **క** *ని అనుసరించి* **వ** *చలించినప్పుడు, ఏదో ఒక సంసాధకము* **స** $\in S$ *చలనలో $\Omega(\log n)$ శ్రమిస్తుంది.*

ఈ లేఖలో అధోబంధాలే కాక ఉద్బంధాలను కూడా పరిశీలిస్తాం. ఆ ఉద్బంధాలను ప్రతిష్టించే విధికల్పాలకు ఉపకరించే ఇంకొక ముఖ్యమైన పరికరం f-పీఠిక.

నిర్వచనం 10.2.4. ఒక n కోష్టాలు గల f**-పీఠిక** అనే వస్తువుకు $\mathbf{క}_1, \ldots, \mathbf{క}_n$ అనే కోష్టాలుంటాయి. పైగ, క్రింది ఉపవిధులను భరిస్తుంది:

1. పీఠికలోని ప్రతి కోష్టము పఠ, లిఖ, పోల్చిమార్చు అనే నియోజ్యాలను సాధారణరీతిలో భరిస్తుంది
2. ప్రతి **ప** అనే f-పీఠిక, ఏదో ఒక ప్రత్యేకమైన నియోజ్యం fతో స్థాపింపబడి ఉంటుంది (ఉదా. $f =$ సంపర్కం, అనగా కూడిక). వస్తువుమీద మీద **ప**.$f()$ ని పిలిస్తే, అది $f(\mathbf{క}_1, \ldots, \mathbf{క}_n)$ను తిరుగిస్తుంది.

లేఖలో f-పీఠికలగురించిన ఒక ముఖ్యమైన సిద్ధాంతాన్ని వాడతాము.

సిద్ధాంతం 10.2.5 (f-పీఠికలు [106]లో Theorem 6)**.** *f అనే నియోజ్యం* **లఘుతమ** *(min),* **బృహత్తమ** *(max), లేదా* **సంపర్క** *(sum) ఐనచో, f-పీఠిక సమస్యకు ప్రతి సాధారణ నియోజ్యం (పఠ, లిఖ, పోల్చిమార్చు) $O(\log n)$ శ్రమతో, f-నియోజ్యమును $O(1)$ శ్రమతో పూర్తిచేసే విధికల్పము కలదు.*

10.3 సామాన్య-జాగృతి-పరిష్కార సమస్య

ఈ భాగంలో జయంతి జాగృతి-పరిష్కారాన్ని సామాన్యపరుస్తాం. మా సామాన్య-జాగృతి-పరిష్కార సమస్య శ్రేణిలోని ప్రతి సమస్యను ఒక అహ్రాసక (దుర్బల వర్ధన) శ్రేధి పరిమితి $s_1, \ldots, s_n$ ద్వారా విశదీకరిస్తాం. సామాన్య-జాగృతి-పరిష్కార సమస్య **జ**$(s_1, \ldots, s_n)$ అనగా, సుతథ్యముగా:

నిర్వచనం 10.3.1 (జాగృతి-పరిష్కారం)**.** ప్రతి వర్ధన శ్రేధి $0 \leq s_1 \leq s_2 \leq \cdots \leq s_n \leq n$ కి సంవాదిగా **జ**$(s_1, \ldots, s_n)$ అనే n సంసాధక సంవిధానపు **సామాన్య-జాగృతి-పరిష్కార** సమస్య ఉంటుంది. ఆసమస్యను పూరించే విధికల్పమునకు మూడు గుణములుండాలి:

1. **సమాప్తి:** ప్రతి సంసాధకము [1, n]లో పూర్ణసంఖ్య తిరుగివ్వాలి.
2. **సత్యవాక్కు:** ఏ సంసాధకము k సంసాధకాలు లేచేముందు, kను తిరుగివ్వకూడదు.
3. **అవితండం:** k, లేదా యెక్కువ, సంసాధకాలు s_k కంటే తక్కువ సంఖ్యను తిరుగివ్వకూడదు.

ఈ శ్రేణిలో **జ**$(1, 1, \ldots, 1, n)$ సమస్యను **సులభ జాగృతి-పరిష్కారం (సులభ సమస్య)** అని **జ**$(1, 2, \ldots, n-1, n)$ సమస్యను **కఠిన జాగృతి-పరిష్కారం (కఠిన సమస్య)** అని వ్యవహరిస్తాం.

సులభ జాగృతి-పరిష్కారము జయంతి జాగృతి-పరిష్కారానికి ప్రతిరూపం, యెందునంటే ఈ రెండు సమస్యలమధ్య సులభతరమైన పరస్పర న్యూనీకరణ సాధ్యము. వివరముగా, **వ** అనే విధికల్పము జయంతి జాగృతి-పరిష్కారాన్ని పూరిస్తే, **వ** ఎక్కడెక్కడ అసత్తును తిరుగిస్తుందో, అక్కడక్కడ 1ని, **వ** ఎక్కడెక్కడ సత్తును తిరిగిస్తుందో, అక్కడక్కడ nను తిరుగిచ్చే **వ**′ అనే విధికల్పము సులభ జాగృతి-పరిష్కారాన్ని పూరిస్తుంది. ప్రతిదిశలో, **వ** అనే విధికపము సులభ జాగృతి-పరిష్కారాన్ని పూరిస్తే, **వ** ఎక్కడెక్కడైతే n కంటే తక్కువ సంఖ్యను తిరుగిస్తుందో, అక్కడక్కడ అసత్తును, **వ** ఎక్కడెక్కడైతే nను తిరుగిస్తుందో, అక్కడక్కడ సత్తును తిరుగిచ్చే **వ**′ సులభ జాగృతి-పరిష్కారాన్ని పూరిస్తుంది. అందుచేత, ఇప్పటినుంచి జయంతి జాగృతి-పరిష్కార సమస్యని కూడా సులభ సమస్యగానే వ్యవహరిస్తాం.

సిద్ధాంతం 10.3.2 (సామాన్య జాగృతిపరిష్కార సిద్ధాంతం)**.** *$\mathbf{స}_1, \ldots, \mathbf{స}_n$ సంసాధకాల మధ్య సామాన్య జాగృతిపరిష్కార సమస్య $\mathbf{జ}(s_1, \ldots, s_n)$ ని **వ** అనే వినా-ప్రతీక్ష విధికల్పం పూరిస్తుందని భావిద్దాం. అప్పుడు నిశ్చయముగా, ఏదోవొక కాలబంధం **క** ప్రకారం **వ** చలిస్తే, చలనకు $\Omega(n + \sum_{i=1}^{n} \log s_i)$ సంశ్రమ ఖర్చౌతుంది.*

నిరూపణ. $\mathbf{జ}(s_1, \ldots, s_n)$ ని **వ** అనే వినా-ప్రతీక్ష విధికల్పం పూరిస్తుందని భావిద్దాం. సిద్ధాంతం 10.2.2 లో ప్రస్తుతింపబడ్డ **క** అనే కాలబంధాన్ని పరీక్షిద్దాం. ఈ కాలబంధ క్రమంలో చలిస్తే ఏ **స** అనే సంసాధకము **త** అనే సంఖ్యను $\log_4$ **త** కల్పాల ముందు తిరుగివ్వలేదని విరోధోక్తి న్యాయం ద్వార నిరూపిద్దాం.

స సంసాధకము **క** కాలబంధంలో అన్ని సంసాధకాలు చలిస్తున్నప్పుడు **ట** $<$ $\log_4$ **త** అనే పూర్వ కల్పంలోనే **త** ను తిరుగిచ్చిందని భావిద్దాం. అవిచక్షణీయత ప్రకారం జ్ఞానగణంలో ఉన్న, అనగా $\mathbf{జ్ఞా}_{\mathbf{ట}}(\mathbf{స})$ లో ఉన్న, సంసాధకాలు మాత్రం **క** క్రమములో చలించినచో **స** సంసాధకము **త** ను **ట** కల్పంలో తిరుగిస్తుంది. కానీ, సిద్ధాంతం 10.2.2 నలగవ అంశం ప్రకారం $|\mathbf{జ్ఞా}_{\mathbf{ట}}(\mathbf{స})| \leq 4^{\mathbf{ట}} < 4^{\log_4 \mathbf{త}} = \mathbf{త}$, అందుచేత ఈ తిరుగిచ్చిన సంఖ్య సత్యవాక్కును వ్యతిరేకిస్తుంది. విరోధోక్తి సంపూర్ణం. తద్ప్రమాణముగ, **త** లేదా ఎక్కువ సంఖ్యను తిరుగిచ్చిన ప్రతి సంసాధకము కనీసం $\log_4$ **త** శ్రమిస్తుందని గుర్తుంచుకుందాం.

అవితండం వలన $\mathbf{జ}(s_1, \ldots, s_n)$ ని పూరించే ప్రతి విధికల్పంలో కనీసం ఒక సంసాధకము s_nకి తక్కువకాని సంఖ్యను, కనీసం ఇంకొక సంసాధకము s_{n-1}కి తక్కువకాని సంఖ్యను, ఇత్యాది. తిరుగివ్వాలి కనక, సంశ్రమ $\geq \sum_{i=1}^{n} \log_4 s_i = \Omega(\sum_{i=1}^{n} \log s_i)$ అని తేలింది. □

ఉపసిద్ధాంతం 10.3.1 (కఠిన సమస్యాధోబంధం)**.** *n సంసాధకాల సంవిధానంలో కఠిన సమస్యను పూరించే ప్రతి విధికల్పము ప్రతికూలతమ చలనలో $\Omega(n \log n)$ సంశ్రమిస్తుంది.*

ఉపసిద్ధాంతం 10.3.2 (సులభ సమస్యాధోబంధం)**.** *n సంసాధకాల సంవిధానంలో సులభ సమస్యను పూరించే ప్రతి విధికల్పము ప్రతికూలతమ చలనలో $\Omega(n)$ సంశ్రమిస్తుంది.*

పై నిరూపించిన అధోబంధాలను ఆసరాగా తీసుకొని న్యూనీకరణలద్వారా కొన్ని చేక్కటి క్రొత్త ఫలాలను నిరూపిస్తాం.

10.4 అధోబంధాలు

సమకాలిక కృత్రిమాణు దత్తాంశసంవిధానాల శ్రమవిషమతను తగ్గించే దిశలో ఈ మధ్య చేలా పని జరిగింది. ఈ ప్రగతిలో రెండు కీలకోదాహరణాలు జయంతియొక్క గణత్రము [106], ఎలెన్ వోల్ఫల్‌ల తెచ్చిమార్చు వస్తువు [57]. ఇరువస్తువులు ప్రతికూలతమశ్రమలో సర్వోత్కృష్టులని జయంతి అధోబంధాన్నిబట్టి తేలుతున్నది [103]. కాని, సర్వసాధారణముగ ఒక్కటే వస్తు ఉపవిధిని పిలవడంకన్న చేలా వస్తూపవిధులను పిలవడం సంభవిస్తుంది. అందుకని ఒక్క ఉపవిధియొక్క ప్రతికూలతమశ్రమతో పాటు (ప్రతికూలతమ) సంశ్రమను గణించేడము ముఖ్యము. పై పేర్కొన్న రెండు వస్తువులు సంశ్రమవిషమతలో సర్వోత్కృష్టులని మా సామాన్య జాగృతిపరిష్కార అధోబంధం ద్వారా మొదట్టిసారిగా ఈ లేఖలో ప్రదర్శిస్తాం.

పరిశోధనలో అల్పవిషమత గల దత్తాంశసంవిధానాలను తయారు చేయలేని పక్షాలలో, సమీప-దత్తాంశసంవిధానాలు (తక్కువ ఆవశ్యకతలు పాటించే దత్తాంశసంవిధానాలు) సృజింపబడ్డాయి. ఉదాహరణకు, h-సమీప ప్రాధాన్యతాపంక్తి అనగా ప్రాధాన్యతాపంక్తిలోని న్యూనతమనిష్కాస ఉపవిధిలో, న్యూనతమ ప్రాధాన్యతగల సంఖ్యేగాక h న్యూనతమ ప్రాధాన్యతలుగల సంఖ్యలలో ఏదైనా తిరుగివ్వచ్చు. ఆధునికముగ, న్యూనతమనిష్కాస ని $O(\log^3 n)$ అపేక్షిత అడుగులలో నిర్వహించే స్ప్రేలిస్ట్ అనే $O(n \log^3 n)$-సమీప ప్రాధాన్యతాపంక్తిని ఆలిస్టార్ ఇత్యాదులు కల్పించారు [4, 5]. ఈ లేఖాభాగంలో అటువంటి సమీప-దత్తాంశసంవిధానాలనుదేశించే అధోబంధాలను కూడ ప్రదర్శిస్తాము.

సమీప-దత్తాంశసంవిధానలగురించిన సిద్ధాంతాలను వ్యక్తీకరించేడానికి క్రింది పరిభాషానిర్వచనాలు ఉపకరిస్తాయి.

నిర్వచనం 10.4.1 (సమీప దత్తాంశసంవిధానాలు)**.**

1. **h-సమీప గణత్రం**కి సామాన్య గణత్రమువోలెనే 0-విలువతో మొదలుగొని పెంచే$()$, విలువ$()$ అనే ఉపవిధులను భరిస్తుంది. కానీ, విలువను తిరుగిచ్చునపుడు అసలు విలువకు h వరకు యెక్కువైనా, తక్కువైనా గల సంఖ్యను తిరుగివ్వవచ్చే.

2. **h-సమీప పంక్తి, రాశి, ప్రాధాన్యతాపంక్తి**లు ఆ ఆ సామన్య వస్తువులువోలే ఉంటాయి. కానీ, క్రమములో మొదట్టి సంఖ్యను నిశ్చయముగ తిరుగివ్వకుండ, క్రమములో మొదటి h సంఖ్యలలో ఒకటి తిరుగిస్తాయి.

సిద్ధాంతం 10.4.2 (గణత్ర అధోబంధాలు)**.** *గణత్రసదృశ వస్తువులు గురించిన కొన్ని సత్యాలు క్రింది ఇవ్వబడ్డాయి:*

1. *ఆ అనే విధికల్పము n సంసాధకాలకు కృత్రిమాణు తెచ్చిపెంచు వస్తువని భావిద్దాం. ప్రతి $m \geq n$కి ఏదో ఒక m ఉపవిధుల అనుక్రమము $\Omega(m \log n)$ సంశ్రమను (ప్రతికూలతమములో) కలిగిస్తుంది.*

2. ***గ** అనే విధికల్పము n సంసాధకాలకు కృత్రిమాణు గణత్రమని భావిద్దాం. ప్రతి $m \geq n$కి ఏదో ఒక $2m$ ఉపవిధుల అనుక్రమము $\Omega(m \log n)$ సంశ్రమను (ప్రతికూలతమములో) కలిగిస్తుంది.*

3. *గ అనే విధికల్పము n సంసాధకాలకు కృత్రిమాణు $(1-\varepsilon)n/2$-సమీప గణత్రమని భావిద్దాం; యత్ర $\varepsilon > 0$ ఐన అచలం. ప్రతి $m \geq n$కి ఏదో ఒక $2m$ ఉపవిధుల అనుక్రమము $\Omega(m \log n)$ సంశ్రమను (ప్రతికూలతమములో) కలిగిస్తుంది.*

నిరూపణ. వాక్యాలను క్రమముగా నిరూపిస్తాం.

1. **గ** తొలి స్థితిలో 1 విలువతో ఉంటుంది. $m = n$ ఐతే, ప్రతి సంసాధకము తెచ్చిపెంచే() ను పిలిచి, వచ్చిన తిరుగువిలువను తిరుగిస్తే, కఠిన సమస్యకు పూరణౌతుంది. అందున, నిరూపణ సిద్ధాంతం 10.3.1 ద్వారా సమాప్తము. వేరొక పూర్ణసంఖ్య kకు $m = kn$ ఐతే, పై ఉపాయమును కల్పాలలో పునస్కరించచ్చు. అనగా, కల్పాలచివరల తడులు పెట్టి ఆగుతూ, ప్రతి కల్పములో కఠిన సమస్యను పూరిస్తే k మాట్లు కఠిన సమస్యను పూరించినందుకు $\Omega(kn \log n) = \Omega(m \log n)$ సంశ్రమ తథ్యం. ఒకటే జాగ్రత్తేమిటంటే, $h^{\text{వ}}$ మాటు పునస్కరించేనప్పుడు విలువనుంచి $(h-1)n$ను తీసివేసి $[1, n]$లో సంఖ్యను తిరుగివ్వాలి. (m అనే సంఖ్య n అనే సంఖ్యచేత భాగింపబడకపోతే, $n\times$లబ్ధం మాట్లు పై మార్గమును అనుసరించి, శేషము మాట్లు ఉట్టినే ఉపవిధిని పిలిస్తే సరిపోతుంది.)

2. ప్రతి సంసాధకము **గ**.పెంచే()ని పిలిచి, తరువాత **గ**.విలువ()ను తిరుగిస్తే కఠిన సమస్య పూరింపబడుతుంది. (ఈ విషయాన్ని విధికల్పం 30 ని ప్రస్తుతించినప్పుడు వివరిస్తాం.) అందుచేత $m = n$ ఐతే నిరూపణ సిద్ధాంతం 10.3.1 ద్వారా సమాప్తము. k అనే పూర్ణ సంఖ్యకు $m = kn$ ఐతే, పై యుక్తిని k మాట్లు, మధ్యన తడులు పెట్టి ఆగుతూ, అవలంబిస్తే సరిపోతుంది. ఒకటే జాగ్రత్తేమిటంటే, $h^{\text{వ}}$ మాటు అవలంబించేనప్పుడు విలువనుంచి $(h-1)n$ను తీసివేసి $[1, n]$లో సంఖ్యను తిరుగివ్వాలి.

3. ముందు $m = n$ అని భావించి పరిశీలిద్దాం. సమీప-గణత్రమునకు కూడా విధికల్పములో మొదట్ట **గ**.పెంచే()ని పిలవడం, తరువాత **గ**.విలువ()ను పిలవడం ఉంటాయి. కానీ వచ్చిన సంఖ్యను సరాసరి తిరుగివ్వకుండా, రాబోయే మార్పు చేస్తాము. తిరిగొచ్చిన విలువ **త** ఐనచో, తప్పకుండ కనీసం, **త** $-(1-\varepsilon)n/2$ సంసాధకాలు లేచాయని నిర్ధారించేచ్చు. అందున, ఆ సంఖ్యను (ఆ సంఖ్య ఒకటికంటే తక్కువైతే 1ని) తిరుగిస్తాం. మొదటి $n/2 + (1-\varepsilon)n/2 = (1-\varepsilon/2)n$ సంసాధకాలు **గ**ను పెంచిన తరువాత, **గ** యొక్క అసలు విలువ $(1-\varepsilon/2)n$ను దాటి వుంటుంది. అందుకని చివరి $\varepsilon n/2$ సంసాధకాలు సమీప-గణత్రమైననూ $n/2$ను మించిన విలువలను చూసి, $n/2 - (1-\varepsilon)n/2 = \varepsilon n/2$ ను మించిన విలువలను తిరుగిస్తాయి. అందుకని, ఈ వివరించిన విధికల్పము $s_1 = 1, \ldots, s_{n-\varepsilon n/2} = 1, s_{n-\varepsilon n/2+1} = \frac{\varepsilon}{2}n, \ldots, s_n = \frac{\varepsilon}{2}n$ పరిమితితో గల **జ**$(s_1, \ldots, s_n)$ సమస్యను పూరిస్తుంది. అందుచేత $\sum_{i=1}^{n} \log s_i = \varepsilon n/2 \log(\varepsilon n/2) = \Omega(n \log n) = \Omega(m \log n)$ సంశ్రమ వ్యయపరుస్తుంది (ప్రతికూలతమ చలనలో).

   $m = kn$ ఐనచో, కల్పాలలో పునస్కరిస్తే సరిపోతుంది.

□

ఉపవాక్యం. *సిద్ధాంతం 10.4.2 ప్రకారం జయంతి గణత్రము [106], ఎలెన్ వోల్ఫల్ల తెచ్చిపెంచే వస్తువు [57] సంశ్రమవిషమతలో సర్వోత్కృష్టలు.*

సిద్ధాంతం 10.4.3. **వ** *అనే విధికల్పము కృత్రిమాణు h-సమీప పంక్తి కాని, రాశి కాని, ప్రాధాన్యతాపంక్తి కాని కానివ్వండి; యత్ర $h = (1-\varepsilon)n$ మరియు $\varepsilon > 0$ ఐన అచలం. ప్రతి $m \geq n$కి, ఏదో ఒక కాలబంధములో m నిష్కాసనోపవిధుల అనుక్రమము $\Omega(m \log n)$ సంశ్రమను వ్యయపరుస్తుంది.*

నిరూపణ. ముందు **వ** అనే వస్తువు పంక్తని భావిద్దాం; అలాగే $m = n$ అని భావిద్దాం. తొలి స్థితిలో **వ**లో క్రమముగా 1,2,...,n ని ఉంచేదాం. అప్పుడు ప్రతి సంసాధకము ఒక మాటు నిష్కాసిస్తే, దానికి ఏదో ఒక సంఖ్య లభిస్తుంది. కనీసం $\frac{\varepsilon}{2}n$ సంసాధకాలు నిష్కాసించే దాక h-సమీపత ప్రకారం, ఏ సంసాధకానికి $\frac{1-\varepsilon}{2}n$ ను మించిన సంఖ్య తిరుగురాదు. అలాగే ప్రతి సంసాధకానికి ఒక సంఖ్య వస్తుంది కాబట్టి, కనీసం $\frac{\varepsilon}{2}n$ సంసాధకాలకు $\frac{1-\varepsilon}{2}n$ ను మించిన సంఖ్య వస్తుంది. అందుచేత, నిష్కాసించినప్పుడు $\frac{1-\varepsilon}{2}n$ ను మించిన తిరుగు-సంఖ్య వచ్చిన ప్రతి సంసాధకము $\frac{\varepsilon}{2}n$ను తిరుగిచ్చి, ఇతర సంసాధకాలన్నియు 1ని తిరుగిచ్చే విధికల్పము $s_1 = 1, \ldots, s_{n-\varepsilon n/2} = 1, s_{n-\varepsilon n/2+1} = \frac{\varepsilon}{2}n, \ldots, s_n = \frac{\varepsilon}{2}n$ పరిమితితో గల **జ**$(s_1, \ldots, s_n)$ సమస్యను పూరిస్తుంది. అందుచేత $\sum_{i=1}^{n} \log s_i = \varepsilon n/2 \log(\varepsilon n/2) = \Omega(n \log n) = \Omega(m \log n)$ సంశ్రమ వ్యయపరుస్తుంది (ప్రతికూలతమ చలనలో). ఎప్పటిలాగే, $m > n$ ఐనచో, కల్పాలలో పునస్కరిస్తే సరిపోతుంది.

ప్రాధాన్యతాపంక్తులకు ప్రాధాన్యతలు సంఖ్యలు ఒకటే అయ్యేడట్టు పెడితే ఇదే నిరూపణ సరిపోతుంది. రాశులకి క్రమమును తిరగేస్తే సరిపోతుంది. □

ఉపవాక్యం. *పై ప్రదర్శించిన దత్తాంశసంవిధాన అధోబంధాలతో సైతం, ప్రస్తుతపు రచయిత జయంతి ఇత్యాద్యులు గోష్టిపత్రములో గణ-సంధి (set union) అనే వస్తువుకు కడు దుష్కరమైన న్యూనీకరణాధోబంధాన్ని స్థాపించేరు [117]. అలాగే, ఆ పత్రములో సూచింపబడ్డ కొన్ని సిద్ధాంతాలకు నిరూపణలు ఈ భాగంలో ఇప్పటికే విస్తృతింపబడి ఉన్నాయి.*

10.5 ఉద్బంధాలు

10.5.1 సులభ సమస్య

ఈ లేఖలో మొట్టమొదటి క్రొత్త ఉద్బంధ-యోగదానముగా సులభ సమస్యకు అల్పసంశ్రమ విధికల్పమిస్తున్నాము. అవగమనార్థం, m అనే పూర్ణసంఖ్యకు $n = 2^m$ అని భావిద్దాం. మా విధికల్పము (సంవిభక్త)స్మృతిలో ఒక n పర్ణులు గల $2n-1$ గ్రంథులు గల సమగ్ర ద్విమయ వృక్షాన్ని స్థాపించి, ప్రతి గ్రంథిలో 0సంఖ్యను తొలిస్థితిలో వ్రాసి ఉంచేతుంది. ఆ పిమ్మట, **స**$_i$ సంసాధకము, క్రింది ఉపవిధిని $k = 1$, $x = i^{\text{వ}}$ పర్ణం అనే పరిమితులతో నిర్వహిస్తుంది:

విధికల్పం 29 సరళ సమస్యను పూరించే విధికల్పములో వృక్షోపవిధి.

ఉపవిధి వృక్షోపవిధి(k, x)
  యదీ పోల్చిమార్చు$(x, 0, k)$
    యదీ x మూలం **తర్వా తిరుగివ్వు** $2k$
    అన్యథా యదీ x.సోదర $\neq 0$ **తర్వా తిరుగివ్వు** వృక్షోపవిధి$(2k, x$.పితృ$)$
  అన్యథా తిరుగివ్వు k

సిద్ధాంతం 10.5.1. *వృక్షోపవిధి సరళ సమస్యను $O(n)$ సంశ్రమతో పూరిస్తుంది.*

నిరూపణ. ముందు వృక్షోపవిధి సరళ సమస్యను పూరిస్తుందని ప్రతిష్టిస్తాం. సరళ సమస్య సమాధానానికి మూడు గూణాలు. ప్రతి గుణాన్ని విడిగా నిరూపిస్తాం.

1. **సమాప్తి:** ప్రతి సంసాధకము దర్శించిన ప్రతి గ్రంథి వద్ద $O(1)$ శ్రమ మాత్రం ఖర్చుచేస్తుంది. ప్రతి సంసాధకము అధికతమముగా వృక్షోన్నతి, అనగా $\log n$, గ్రంథులని దర్శిస్తుంది కాబట్టి, సమాప్తి నిశ్చయం.

2. **సత్యవాక్కు:** ప్రతిష్టాపన ద్వారా h ఉన్నతిలో ఉన్న **గ** గ్రంథిలో 0 లేకపోతే, ఆ గ్రంథిక్రిందున్న పర్ణాలలో మొదలైన 2^h సంసాధకాలు లేచాయని నిరూపిస్తాం. విధికల్పం తొలిస్థితిలో అన్ని గ్రంథుల్లో 0 ఉంటుందికాబట్టి ప్రతిష్టాపన మూలం నిలుస్తుంది. ప్రతి సంసాధకము **స**$_i$ వేరే పర్ణం **ప**$_i$తో మొదలిడుతుంది కాబట్టి, ప్రతి పర్ణం **ప**$_i$ దగ్గర ప్రతిష్టాపన నిలుస్తుంది. సంసాధకములు ఇరు పుత్రగ్రంథులు 0 కాదని స్థాపించేకున్న తరవాతే పితృ గ్రంథిని 0-కాని సంఖ్యకు పోల్చిమార్చడానికి ప్రయత్నిస్తాయి కాబట్టి ప్రతిష్టాపనపదం నిలుస్తుంది.

   సంసాధకము h ఉన్నతికి చేరిన తరువాతే $k = 2^h$ తిరుగిస్తుంది కాబట్టి, సత్యవాక్కు నిశ్చయం.

3. **అవితండం:** ప్రతిష్టాపన ద్వారా ప్రతీ గ్రంథి xని ఏదో ఒక సంసాధకం పోల్చిమార్చడానికి ప్రయత్నిస్తుందని నిరూపిస్తాం. ప్రతీ పర్ణము ఒక సంసాధక మూల స్థానము కాబట్టి ప్రతీ పర్ణమువద్ద ప్రతిష్టాపన నిలుస్తుంది. x పర్ణేతర గ్రంథైతే, దాని పుత్రగ్రంథులమీద పోల్చిమార్పులు జరిపిన సంసాధకాలలో చివర పోల్చిమార్పులో ఉత్తిర్ణమైన సంసాధకమును **స**$_i$ అని, పోల్చిమార్చిన పుత్రగ్రంథిని **గ**అని అన్దాం. ఆ సంసాధకము **గ**.సోదర $\neq 0$ అనే పరీక్ష చేసినప్పుడు సత్తు వస్తుంది. కావున ప్రతిష్టాపనపదం నిలుస్తుంది.

   వృక్షమూలము కూడ ఒక గ్రంథి కాబట్టి కనీసం ఒక సంసాధకమైన మూలాన్ని పోల్చిమార్చడానికి ప్రయత్నిస్తుంది. మూలాన్ని మార్చిన ప్రతి సంసాధకము n ని తిరుగిస్తుంది. అందున అవితండం నిశ్చయం

శ్రమవిషమతను నిరూపించేడం జ్యామితిక శ్రేఢి సంపర్కము ద్వారా శులభం. ప్రతి గ్రంథిని ఏదో ఒక సంసాధకము మాత్రము పోల్చిమార్చగలదు. పోల్చిమార్పును ప్రయత్నించి ఉత్తీర్ణమవ్వని సంసాధకము వెంటనే తిరుగిస్తుంది. కాబట్టి, పోల్చిమార్పు ప్రయత్నాలు గ్రంథులసంఖ్యకు సంసాధకాలసంఖ్యను కలిపినకంటే ఉండలేవు. కావున,

పోల్చిమార్పు ప్రయత్నాలు అధికతమముగ $2n - 1 + n = 3n - 1$. ఉపవిధిని పిలిచిన ప్రతిసారి పోల్చిమార్పు ప్రయత్నమౌతుంది కాబట్టి, సంశ్రమ $O(n)$.

□

పై సిద్ధాంతాన్ని సులభ సమస్యాధోబంధంతో జోడించిన, వచ్చే ఉపసిద్ధాంతం మిగుల సుళువుగా ఉత్పన్నమౌతుంది:

ఉపసిద్ధాంతం 10.5.1. *సులభ సమస్యయొక్క సంశ్రమ* $\Theta(n)$.

10.5.2 కఠిన సమస్య

సులభ సమస్యను సమూహఖేలన-వృక్షోపవిధిద్వారా తక్కువ సంశ్రమతో పూరించేగలిగాము. ఇప్పుడు కఠిన సమస్య మీదకు దృష్టి సారిద్దాం. ఈ సమస్యకు ఇథఃపూర్వం నిరూపించిన సిధాంతప్రకారం అధోబంధం $\Omega(n \log n)$. ఈ సమస్యను సర్వోత్తమ సంశ్రమతో, అనగా $O(n \log n)$ సంశ్రమతో, పూరించచ్చేని f-పీఠికల ద్వార విధికల్ప ప్రమాణముగా నిరూపిస్తాం.

ఊహ సులభమైనదే. $f =$ సంపర్కం గా కల **గ** అనే f-పీఠికను పెట్టుకోని, క్రింది విధికల్పాన్ని ప్రతి సంసాధకము నిర్వహిస్తే చాలు:

విధికల్పం 30 కఠిన సమస్యను పూరించే విధికల్పములో గణిత్రోపవిధి.

ఉపవిధి గణిత్రోపవిధి()
  గ.పెంచే()
  తిరుగివ్వు **గ**.విలువ()

సిద్ధాంతం 10.5.2. *గణిత్రోపవిధి కఠిన సమస్యను* $O(n \log n)$ *సంశ్రమతో పూరిస్తుంది.*

నిరూపణ. **గ** కృత్రిమాణువస్తువు కనక సమాప్తి నిశ్చయం. **గ**ని ప్రతి సంసాధకము ఒక్కమాటే పెంచేతుంది కనక **గ** విలువ ఎప్పుడు లేచిన సంసాధక సంఖ్యను అధిగమించేదు; దాంతో సత్యవాక్కు నిశ్చయం. $k^{\text{వ}}$ స్థానములో మొదటి పంక్తిని పూర్తిచేసిన (రేఖింపబడ్డ) సంసాధకము కనీసం kని తిరుగిస్తుంది; అందున అవితండము నిశ్చయం.

f-పీఠికల పూర్వాంశ సిద్ధాంత ప్రకారం ప్రతి సంసాధకము విషమతమ చలనలోకూడా $O(\log n + 1) = O(\log n)$ శ్రమిస్తుంది. అందున, సంశ్రమ $O(n \log n)$. □

పై నిరూపించేన ఉద్బంధము అధోబంధముతో కలుస్తుంది. అందుచేత క్రింది దృఢ పరిబంధం ఉత్పన్నమౌతుంది:

ఉపసిద్ధాంతం 10.5.2. *కఠిన సమస్యయొక్క సంశ్రమ* $\Theta(n \log n)$.

10.6 సమాప్తి

ఈ లేఖలో కృత్రిమాణుదత్తాంశసంవిధానాల సంశ్రమ విషమతను నిరూపించేటకు ఉపకరించే సామాన్య జాగృతి-పరిష్కారమనే సమస్యాశ్రేణిని నిర్వచించి పరిశీలించాము. పరిశీలనలో ముఖ్యభాగాలుమూడు, (1) శ్రేణిలోని ప్రతి సమస్యకు ఒక అధోబంధాన్ని నిరూపించేడం, (2) న్యూనీకరణల ద్వారా గణత్రము, పంక్తి, రాశి, ప్రాధాన్యతాపంక్తి వంటి దత్తాంశసంవిధానాలకు అధోబంధాలను స్థాపించేడం, (3) జాగృతి-పరిష్కార కఠిన సమస్యకు ఉద్బంధమును నిరూపించి, ఆ ద్వారా ఈ పద్ధతికి గల పరిమితులను కూడా ప్రదర్శించేడం. మా జాగృతి పరిష్కార విధానము ద్వారా తొలి సారిగా, ఎలెన్వోల్ఫల్ల తెచ్చిపెంచే దత్తాంశము [57], అలాగే జయంతి గణత్రము [106] సంశ్రమ సర్వోత్కృష్టులని నిరూపించేగలిగాము. మా పద్ధతి ఈ ప్రస్తుత లేఖలోని సిద్ధాంతాలకేగాక, జయంత్యాదుల చేత మరింత దుష్కరమైన గణ-సంధి అధోబంధానికి [117] దారితీయడం గమనార్హం. గణ-సంధికి ఇంకా దృఢమైన అధోబంధాన్ని ఇవ్వడం (లేదా ఉద్బంధాన్ని దృఢపరచడం) కృత్రిమాణువస్తువులలో ఒక ముఖ్య ఉత్ఖాటిత సమస్యగా నిలచివుంది. జాగృతి-పరిష్కరణ ద్వారా ఇతర వస్తువులకు అధోబంధాలను నిరూపించేడము కూడా ఆకర్షణీయమైన దిశే.

శిక్షణాత్మకముగా చూసినచే, మా ఈ లేఖ తెలుగుభాషలో తొలి ఆధునిక సంగణక శాస్త్ర పరిశోధన పత్రికని మా నమ్మకము. కానీ, సంస్కృతములో విలువైన గణిత, సంగణక శాస్త్ర సంపదగలదని ప్రసిద్ధము. ఆ సంప్రదాయమే తెలుగు శాస్త్రమునకు, ప్రత్యేకముగా ఈ లేఖకు స్ఫూర్తి. తదానుసారం, లేఖలోని పారిభాషికపదాలు చాలా మటుకు సంస్కృతరూపాంతరాలు; అందున సరళముగా సంస్కృతములోకి, ఇతర భారతీయభాషలలోకి, అలాగే భారతేతరభాషలలోకి అనువదింపశక్యముగా వుండునని మా ఊహ. తెలుగుభాషలోను, సంస్కృతభాషలోను నూతన ఆధునిక శాస్త్ర పూరోగమన లేఖలు ఇత్యాదిగా కొనసాగాలని, ఆ లేఖల వలన ఎందరో స్ఫూర్తినీ శాస్త్ర జ్ఞానమును పొందగలరని, ఆత్మస్థైర్యాన్ని పెంపొందిచ్చేకొనగలరని మా అభిలాష.

Part IV

Machine Verification

Chapter 11

A Univeral, Sound, and Complete Technique for Machine-Verifiable Proofs of Linearizability

11.1 Introduction

Data structures that organize, store, and quickly recall important pieces of information are the fundamental building-blocks behind fast algorithms. Thus, efficient and *rigorously proved* data structures are fundamental to reliable algorithm design. The task of designing such data structures for shared-memory multiprocessors, however, is notoriously difficult. Due to asynchrony, a t step algorithm for even just two processes has 2^t, i.e. exponentially many, possible executions depending on how the steps of the processes interleave. In fact, even deterministic concurrent algorithms have uncountably many possible infinite executions, as opposed to the single possible execution of a deterministic sequential algorithm. Designing algorithms that are correct in all of these executions is a grueling task, and thus, even mission critical concurrent code often suffers from subtle race conditions. For example, a subtle priority inversion bug in its concurrent code crashed the Pathfinder Rover days after its deployment on Mars and jeopardized the entire multi-million dollar NASA space mission [124]. Examples of errors in published concurrent data structures are also not left wanting [37, 53].

11.1.1 Understanding the Problem

Rigorously proving the correctness of a concurrent data structure implementation $\mathcal{O}$ consists of two steps:

1. A *prover*, often the algorithm designer, deeply studies the implementation $\mathcal{O}$ and produces a *proposed proof* P.

2. A *verifier* evaluates the proposed proof P by confirming that each claim made in the proof is mathematically justified, and that the resultant chain of reasoning constitutes a legitimate proof of $\mathcal{O}$'s correctness.

In general, proving any type of correctness of any algorithm can be inherently intellectually difficult and time consuming, since it requires the prover to contemplate the algorithm deeply, understand why it is correct, and express this understanding in the methodical language of mathematics. Verifying many types of proofs can be rather easier, since it is in its essence a mechanical check.

In this work, we focus on concurrent data structures for asynchronous shared-memory multiprocessors. In particular, *linearizability* [93], which states that data structure operations must appear to take place atomically even in the face of tremendous concurrency, has been the longstanding gold standard for concurrent data structure correctness. Its close cousin *strong linearizability* [80], which ensures that even the hyper-properties of the data structure match those of an atomic object has also garnered a lot of recent interest [16].

When the correctness condition being proved is linearizability, we observe, in practice, that even the verification step can be taxing and time-consuming. Firstly, the proposed proof P can often run for several tens of pages of a dense research paper. For example, the original paper on linearizability [93] contains a short seven line implementation of a queue. To prove its linearizability, the authors propose an eighteen line invariant. The proof of correctness of this invariant and its entailment of the queue's linearizability are presented in a 14 page technical report. Those familiar with this queue implementation know that it, like many linearizable algorithms is inherently "very tricky" to prove correct, and thus it is commendable that the authors, Herlihy and Wing, were able to design this algorithm and give a proof of it. Nevertheless, the subtlety and sheer length of the proof P, makes the job of a conscientious verifier quite hard.

In general, depending on the length of an algorithm and its proof, the mere process of verifying the mathematical validity of the linearizability proof can take hours, days, or even weeks. In some cases, due to the difficulty inherent to writing such long and intricate proofs, provers resort to

making high-level "hand-wavy" arguments, or omit proofs altogether. This makes the verifier's job more difficult to impossible. In other cases, the verifier may be a conference reviewer who does not have the time to verify a long proof and thus must skim the details of the proof which may contain errors; or the end-user of the algorithm may not be a verifier, but an engineer who does not have the time or the mathematical preparation to independently verify the algorithm's correctness before using it in a deployed system.

For all of these reasons, it is not shocking that mistakes in concurrent algorithms are so prevalent, even in mission critical deployed code. To avoid such critical errors, we propose designing concurrent data structures whose correctness is *machine-verified* to limit the scope for human error.

11.1.2 Our Work

Informally, an object $\mathcal{O}$ is linearizable, if *for all* finite runs R of any algorithm $\mathcal{A}$ that uses $\mathcal{O}$, and every operation *op* that is performed on $\mathcal{O}$ in the run R, *there exists* a point in time between *op*'s invocation and return where it "appears to take place instantaneously". This definition, in particular the non-constructive existential quantifier "*there exists*" inside the universal quantification, makes it difficult to prove linearizability. This difficulty is only exacerbated, if the proof is to be provided in a way simple enough for a machine to verify. In fact, if approached naïvely, the prover would need to map each run of the algorithm to a linearization, i.e. a description of where in its invocation-response time interval each operation "appears to take place instantaneously", and then prove that each such mapping is legitimate. This is a difficult task, given that it is known that proving even a single fixed run R linearizable is NP-hard [74].

Our goal however, is to devise a method for proving linearizability that not only works for a single implementation, or even a single type, but to devise a method that is *universal* and *complete*. By universal, we mean that our method should be powerful enough to allow for a proof of linearizability for implementations of *any* object type. By complete, we mean that *any linearizable implementation*, regardless of how complex its expression or linearization structure, must be provable by our method. Of course, our method will also be *sound*, meaning that any argument that is given using our method is indeed a correct mathematical proof of linearizability. Finally, we ensure that our method enables machine verifiable proofs by currently available proof assistants, which are generally built to verify proofs of simple program invariants.

Our Contributions

1. We develop a rigorous *universal*, *sound*, and *complete* method for proving linearizability. In particular, we define a universal transformation that takes an arbitrary implementation $\mathcal{O}$ of an arbitrary type τ, and outputs an algorithm $\mathcal{A}^*$, called the *tracker*, and a simple invariant $\mathcal{I}^*$, and prove a theorem that:

   $\mathcal{O}$ is a linearizable implementation of type τ *if and only if* $\mathcal{I}^*$ is an invariant of $\mathcal{A}^*$.

   (Thus, we can produce a machine verified proof that $\mathcal{I}^*$ is an invariant of $\mathcal{A}^*$ to establish that $\mathcal{O}$ is linearizable.)

2. In fact, we give a whole family of transformations that each output different algorithms $\mathcal{A}'$, called *partial trackers*, with associated invariants $\mathcal{I}'$, and prove that for any of these partial trackers:

   $\mathcal{O}$ is a linearizable implementation of type τ *if* $\mathcal{I}'$ is an invariant of $\mathcal{A}'$.

3. We develop a rigorous *universal*, *sound*, and *complete* method for proving strong linearizability. In particular, we show that for each partial tracker $\mathcal{A}'$, there is an alternate associated invariant $\mathcal{I}''$, and we prove that:

   $\mathcal{O}$ is strongly linearizable *if and only if* some partial tracker $\mathcal{A}'$ has its associated $\mathcal{I}''$ as an invariant.

4. Finally, we demonstrate the power of our methods by producing machine-verified proofs of linearizability and strong linearizability for some notable data structures. In particular, we prove the linearizability and strong linearizability of the Jayanti-Tarjan union-find object, which is known to be the fastest algorithm for computing connected components on CPUs and GPUs [51, 95]. Our proof is verified by the proof assistant TLAPS (temporal logic of actions proof system) [153], and is publicly available on GitHub[1]. We and our collaborators have also used our method to produce TLAPS-verified publicly available linearizability proofs [211, 210, 94] of (1) the aforementioned Herlihy-Wing queue, which is notorious for being hard to prove correct [178]; and (2) Jayanti's single-writer, single-scanner snapshot algorithm, in which processes play asymmetric roles [105].

[1]The proofs are available at: https://github.com/visveswara/machine-certified-linearizability

11.2 Related Work

There have been innumerable works on the design and analysis of linearizable and strongly linearizable objects over the years. Here, we will focus only on works whose principal aim is to produce *proofs of linearizability*.

Herlihy and Wing's landmark 1990 paper that introduced linearizability also seeded the discussion on how to prove implementations linearizable [93]. In particular, their paper introduced the concept of *possibilities*, i.e. the notion that we can imagine several different orders in which partially completed operations could return in the future, and consider linearizations that are consistent with such possibilities. They expanded on this framework in a related publication [92]. Their initial ideas in these papers saw fruition in their proof of linearizability of the tricky Herlihy-Wing queue implementation.

Using the power of machine proof assistants in order to partially or fully prove the correctness of linearizable algorithms is a more recent advent. The results in these categories can generally be classified into three types: model checking, ad hoc proofs of particular data structures, and more general techniques.

First, there are several works that model check algorithms [30, 145, 207]. Model checking *does not* prove correctness; rather, it mechanically searches through small runs of the algorithm with a few processes, and returns a counter-example if it finds one.

Second, there are semi machine-verified and fully machine-verified proofs of linearizability for several specific data structures. For instance, Doherty et al. proved the linearizability of Scott's famous lock-free queue implementation [54], Gao et al. spent several years proving the correctness of their concurrent open addressing hash table [69], and Colin et al. proved the correctness of a certain list-based set algorithm [38]. Various other algorithms have also been proved linearizble using shape analyses that examine the pointer structures within an object [205, 20, 9]. These works prove the correctness of some important and interesting concurrent algorithms. However, they are aimed at producing proofs of specific algorithms rather than presenting widely applicable techniques.

Third, there are general purpose techniques. Most of these techniques, such as [204, 206], are sound but not complete or universal. That is, they are targeted at showing the linearizability of a limited class of algorithms, rather than any linearizable implementation of any type. To our knowledge, the only previous sound and complete technique is by Schellhorn et al. [178].

This technique mechanizes Herlihy and Wing's original possibilities proof strategy through the technology of observational refinement mappings and backward simulations. The authors give a single example demonstration of their technique—a mechanized proof of the Herlihy-Wing queue verified by the proof assistant KIV. In contrast to Schellhorn et al.'s technique, our method for proving linearizability only requires familiarity with the notion of an invariant.

Dongol and Derrick wrote an extensive survey on machine assisted proofs of (standard) linearizability [55]. To the best of our knowledge, we are the first to introduce a general machine verifiable proof method for strong linearizability.

11.3 Model and Definitions

A *concurrent system* consists of a set of asynchrounous processes, Π, that communicate through operations on a set of shared objects, Ω. Each process has a distinct name and a set of private registers, including a *program counter*. Each object has a distinct name and a *type*, which specifies the operation's supported by the object and how these operations behave, i.e. how each operation changes the object's state and what response it returns. An *algorithm* specifies a program for each process, and an initial state for each object. An algorithm's *execution* proceeds in steps. In a *step*, any one process atomically executes the line pointed to by its program counter. It is common for algorithms to restrict each line in a program to apply at most one operation on a shared object; however, to ensure that our results apply to a wider class of algorithms, we do not impose such a restriction. In an *asynchronous* execution of the algorithm, a (possibly) *adversarial scheduler* decides which process $\pi \in \Pi$ will execute the next step in its algorithm at each discrete time step. We formalize and expound these notions below.

Definition 11.3.1 (object type)**.** An *object type* τ consists of the following *components*:

- a set of *states* Σ that the object can be in.
- a set of *operations* OP that can be invoked on the object.
- for each $op \in OP$, a set of *arguments* ARG_{op} that the operation op can be called with.
- a set of *responses* RES, a.k.a. *return values*.
- a *transition function* $\delta(\sigma, \pi, op, arg)$ that outputs the new state σ' and the return value res that result when the operation op with argument arg is performed by process π while the

object is in state σ. Formally, the transition function is

$$\delta : \Sigma \times \Pi \times \{(op, arg) \mid op \in OP, arg \in ARG_{op}\} \to \Sigma \times RES$$

Remark 11.3.2. Operations that require "no argument" (i.e. *read*), are modeled as taking an argument from a singleton set (i.e. $ARG_{read} = \{\perp\}$). Similarly, operations that return "no result" (i.e. *write*), are modeled as returning the result *ack*.

Remark 11.3.3. Our definition of δ (as a function) can be modified to a relation to allow various generalizations of the concept of object type—types that are non-deterministic, types where not every process is allowed to perform every operation (e.g., single writer snapshot), and types where an operation's behavior can depend on which process executes the operation (e.g., LL/SC).

To exemplify this definition of an object type, we present the formal description of an integer valued register that supports the Read and CAS operations in the figure Object Type 11.3.4.

Object Type 11.3.4 (Read/CAS register)**.** For a Read/CAS register type that stores integer values, we have:

- $\Sigma = \mathcal{Z}_p$
- $OP = \{Read, \text{CAS}\}$
- $ARG_{Read} = \{\perp\}, ARG_{\text{CAS}} = \mathcal{Z}_p \times \mathcal{Z}_p$
- $RES = \mathcal{Z}_p \cup \{true, false\}$
- Transition function δ is defined as follows:
  - $\delta(u, \pi, Read, \perp) = (u, u)$
  - $\delta(u, \pi, \text{CAS}, (x, y)) = \begin{cases} (y, true), & \text{if } x = u \\ (u, false), & \text{otherwise} \end{cases}$

Definition 11.3.5 (algorithm)**.** A *(concurrent) algorithm* is a tuple $(\Pi, \Omega, \mathcal{C}_0)$, where:

- Π is a set of processes, where each process $\pi \in \Pi$ has a program and private registers, including a program counter pc_π which points to the line in the program that should be executed next. A *process's state* at any point in time is described by the values of its private registers.
- Ω is a set of objects, where each object has a type, and is in one of its states at any point in time.

- $\mathcal{C}_0$ is a non-empty set of configurations, called *initial configurations*, where a *configuration* is an assignment of a state to each object $\omega \in \Omega$ and an assignment of values to the private registers of each process $\pi \in \Pi$.

Definition 11.3.6 (step, event, run)**.**

- A *step* of an algorithm is a triple $(C, (\pi, \ell), C')$ such that C is a configuration, π is a process, ℓ is the line of code pointed to by π's program counter in C, and C' is a configuration that results when π executes line ℓ from C.
- The *event* corresponding to a step $(C, (\pi, \ell), C')$ is (π, ℓ), i.e., process π executing line ℓ.
- A *run* of an algorithm is a finite sequence $C_0, (\pi_1, \ell_1), C_1, (\pi_2, \ell_2), C_2, \ldots, (\pi_k, \ell_k), C_k$ or an infinite sequence $C_0, (\pi_1, \ell_1), C_1, (\pi_2, \ell_2), C_2, \ldots$ such that C_0 is an initial configuration and each triple $C_{i-1}, (\pi_i, \ell_i), C_i$ is a step.

11.3.1 Implementation of an Object

Implementing complex objects, such as queues and snapshots, from primitive objects supported by the underlying hardware (registers supporting read, write, CAS etc.) is a central problem in multiprocessor programming. Below, we describe what an implementation entails. Later on, we will define what it means for an implementation to be *correct*, in the sense of linearizability.

Definition 11.3.7 (implementation)**.** An *implementation $\mathcal{O}$ of an object of type τ initialized to state σ_0* for a set of processes Π specifies

- A set of objects Ω called the *base objects* along with their types and initial states.
- A set of procedures $\mathcal{O}.op_\pi(arg)$ for each $\pi \in \Pi$, $op \in \tau.OP$, and $arg \in \tau.ARG_{op}$. The objects accessed in the code of the procedures must all be in Ω.

To execute an operation op with argument arg on the implemented object $\mathcal{O}$, a process π invokes the method $O.op_\pi(arg)$ (and executes the code in the procedure). The value returned by the method is deemed $\mathcal{O}$'s response to this operation invocation.

Notice that this definition only captures the syntactic aspect of an implementation. We will now build up to defining the correctness of an implementation.

Intuitively, the implemented object is correct if it behaves like an atomic object of the same type. To formally capture correctness, we define *behavior*, the notion of an *atomic implementation*, and the correctness condition of *linearizability*.

11.3.2 Behaviors of an Implementation

Consider an object implementation $\mathcal{O}$, and a run R in which processes invoke operations on $\mathcal{O}$, execute the corresponding procedures of $\mathcal{O}$, and receive responses. By the definition of a run, R is an alternating sequence of configurations and events. Some of the events are *invocation events*, i.e. calls to $\mathcal{O}$'s procedures, and some are *response events*, i.e. the execution of return statements of $\mathcal{O}$'s procedures. (Of course, there are other events, such as the execution of other lines between the call and return of a procedure.) We call the subsequence of R that includes only the invocation and response events the *behavior* in R. For example, if $\mathcal{O}$ is an initially empty queue, a behavior can be

$$(\pi_1, \textbf{invoke } enq_{\pi_1}(5)), (\pi_2, \textbf{invoke } deq_{\pi_2}()), (\pi_3, \textbf{invoke } enq_{\pi_3}(7)), (\pi_2, \textbf{response return } 7), (\pi_2, \textbf{invoke } enq_{\pi_2}(9))$$

Every possible behavior of an implementation $\mathcal{O}$ can be generated by the algorithm of Figure 11.3.1, where each process repeatedly chooses an operation non-deterministically, invokes it by calling the corresponding procedure, and executes the procedure until it returns (receives a response). The next definition captures this discussion.

Initial Configurations:

- ω is an object of type τ, in its initial state σ_0.
- Each process $\pi \in \Pi$ is assigned the program $main_\pi()$; i.e. pc_π is initialized to the first line of main.
- Every other private register of each $\pi \in \Pi$ is initialized arbitrarily.

program $main_\pi()$

a: **while** *true* **do** choose any $(op, arg) \in \{(o, a) \mid o \in \tau.OP, a \in \tau.ARG_o\}$ and invoke $\mathcal{O}.op_\pi(arg)$

Figure 11.3.1: *Generator algorithm* $\mathcal{A}(\mathcal{O})$ that generates all behaviors of an implemented object $\mathcal{O}$ of type τ. Code shown for process $\pi \in \Pi$.

Definition 11.3.8 (implementation runs and behaviors)**.** Let $\mathcal{O}$ be an implementation of a type τ initialized to σ_0 for a set Π of processes. We define the *runs* of $\mathcal{O}$ to be the set of all runs of the *generator algorithm*, $\mathcal{A}(\mathcal{O})$, presented in Figure 11.3.1. Let $\mathcal{R}$ be the set of all runs of $\mathcal{O}$. For any run $R \in \mathcal{R}$, we define $behavior(R)$ to be the subsequence of all the invocation and response events in R. The set of all *behaviors* of $\mathcal{O}$ is $\{behavior(R) \mid R \in \mathcal{R}\}$.

11.3.3 The Atomic Implementation

Implementing an object from base objects of other types is often challenging, but implementing an object $\mathcal{O}$ from a base object ω of the same type is trivial: each procedure $\mathcal{O}.op_\pi(arg)$ is implemented simply by executing $\omega.op_\pi(arg)$ and returning the received response. We call this implementation the *atomic implementation*.

Definition 11.3.9 (atomic implementation)**.** The *atomic implementation* of an object $\mathcal{O}$ of type τ, initialized to σ_0, is the implementation presented in Figure 11.3.2. (On line 2, the implementation resets r_π to $\perp$ as soon as it returns the value.)

Base Object: ω is an object of type τ, initialized to state σ_0.
procedure $\mathcal{O}.op_\pi(arg \in \tau.ARG_{op})$ ▷ for each $op \in \tau.OP$
1: $r_\pi \leftarrow \omega.op_\pi(arg)$
2: **return** r_π
$r_\pi \leftarrow \perp$

Figure 11.3.2: Atomic implementation of $\mathcal{O}$ of an object of type τ initialized to state $\sigma_0 \in \tau.\Sigma$.

11.3.4 Linearizability

We are now ready to define *linearizability*. Intuitively, an object implementation is linearizable if it behaves like an atomic object of the same type. Formally:

Definition 11.3.10 (linearizability)**.** For a set Π of processes, let $\mathcal{O}$ be an implementation of an object of type τ initialized to σ_0, and let $\mathcal{O}_{atomic}$ be the atomic implementation of an object of type τ initialized to σ_0. Furthermore, let $\mathcal{R}$ be the set of all runs of $\mathcal{A}(\mathcal{O})$ and $\mathcal{R}_{atomic}$ be the set of all runs of $\mathcal{A}(\mathcal{O}_{atomic})$. We say a run $R_{atomic} \in \mathcal{R}_{atomic}$ is a *linearization* of a run $R \in \mathcal{R}$ if $behavior(R) = behavior(R_{atomic})$. Correspondingly, we say that a run $R \in \mathcal{R}$ is *linearizable* if it has a linearization $R_{atomic} \in \mathcal{R}$; equivalently, if $behavior(R)$ is also a behavior of O_{atomic}. We say that the implementation $\mathcal{O}$ is linearizable if every finite run $R \in \mathcal{R}$ is linearizable. Equivalently, $\mathcal{O}$ is linearizable if every finite behavior of $\mathcal{O}$ is a behavior of $\mathcal{O}_{atomic}$.

11.3.5 Strong Linearizability

In general, a run R of a linearizable implementation can have multiple linearizations. Intuitively, a linearizable object implementation satisfies *strong linearizability* if for any run of the implementation

R the object can "commit to a specific linearization" $\mathcal{L}(R)$, such that the linearization of any extension of the run R is an extension of $\mathcal{L}(R)$.

Definition 11.3.11. For a set Π of processes, let $\mathcal{O}$ be an implementation of an object of type τ initialized to σ_0, and let $\mathcal{O}_{atomic}$ be the atomic implementation of an object of type τ initialized to σ_0. Furthermore, let $\mathcal{R}$ be the set of all runs of $\mathcal{A}(\mathcal{O})$ and $\mathcal{R}_{atomic}$ be the set of all runs of $\mathcal{A}(\mathcal{O}_{atomic})$. An implementation is *strongly linearizable*, if there is a *linearization function* $\mathcal{L} : \mathcal{R} \to \mathcal{R}_{atomic}$ that maps each run R of the implementation to an atomic run $\mathcal{L}(R)$ of the atomic implementation that is a linearization of it, such that if R_{pre} is a prefix of R, then $\mathcal{L}(R_{pre})$ is a prefix of $\mathcal{L}(R)$.

Not all linearizable implementations are strongly linearizable, since, for some implementations, we need to extend a different linearizations of a run R to linearize different extensions of R. This notion of *strong* linearizability is subtle, but has been shown important in preserving hyperproperties of composed algorithms, such as output probability distributions [80, 16].

11.4 Our Proof Technique for Linearizability

Our goal is to devise a scheme by which algorithmists can produce machine verifiable proofs that complex concurrent object implementations are linearizable. To prove linearizability of an implementation $\mathcal{O}$, our technique is to augment $\mathcal{O}$'s code to produce an *augmented implementation* $\overline{\mathcal{O}}$ in such a way, that $\mathcal{O}$ is linearizable if and only if the generator of $\overline{\mathcal{O}}$, i.e. $\mathcal{A}(\overline{\mathcal{O}})$, satisfies a certain simple invariant. In the following, we describe the rules for augmentation, and some specific augmentations of interest.

11.4.1 Augmenting an Implementation

Informally, an *augmented implementation* $\overline{\mathcal{O}}$ has all the code of $\mathcal{O}$, and more. In particular, $\overline{\mathcal{O}}$ may employ additional *auxiliary* base objects, and at each line manipulate these auxiliary objects along with performing the code of that line in the original implementation.

Definition 11.4.1 (augmentation)**.** Let $\mathcal{O}$ be an implementation of type τ with initial state σ_0 from base objects Ω for processes Π. An *augmentation* $\overline{\mathcal{O}}$ of $\mathcal{O}$ is also an implementation of type τ with initial state σ_0 for processes Π, with the following properties:

- $\overline{\mathcal{O}}$ may employ an additional set of *auxiliary* base objects Ω_{aux}; thus, $\overline{\mathcal{O}}$ is an implementation from $\overline{\Omega} = \Omega \cup \Omega_{aux}$.

- For each line ℓ of each procedure $\mathcal{O}.op_\pi(arg)$, the augmented procedure has a bijectively corresponding line $\overline{\ell}$ of $\overline{\mathcal{O}}.op_\pi(arg)$, such that; $\overline{\ell}$ contains all the code of the corresponding line ℓ and, optionally, some additional code that only changes objects in Ω_{aux}.

Since the additional code has no impact on the original base objects of $\mathcal{O}$ or the private registers of any $\pi \in \Pi$ (including pc_π), we note that $\overline{\mathcal{O}}$ and $\mathcal{O}$ have identical behaviors as summarized below.

Observation 2. If $\overline{\mathcal{O}}$ is an augmentation of $\mathcal{O}$, then the set of behaviors of $\overline{\mathcal{O}}$ is identical to the set of behaviors of $\mathcal{O}$.

When we have two implementations $\mathcal{O}$ and $\overline{\mathcal{O}}$, we will often be interested in analyzing *coupled runs* of these implementations as defined below:

Definition 11.4.2 (coupled runs)**.** Let $\mathcal{O}$ and $\overline{\mathcal{O}}$ be implementations of the same object type τ for the same set of processes Π, with bijectively corresponding lines. We say a run $R_k = C_0, (\pi_1, \ell_1), C_1, (\pi_2, \ell_2), \ldots, C_k$ of $\mathcal{A}(\mathcal{O})$ and a run $\overline{R}_k = \overline{C}_0, (\pi_1, \overline{\ell}_1), \overline{C}_1, (\pi_2, \overline{\ell}_2), \ldots, \overline{C}_k$ of $\mathcal{A}(\overline{\mathcal{O}})$ are coupled, if C_0 and $\overline{C}_0$ share an initialization for all their common variables, and each ℓ_i and $\overline{\ell}_i$ are corresponding lines for each $1 \leq i \leq k$.

11.4.2 The Full Tracker Augmentation

In this section, we consider an implementation $\mathcal{O}$ of an object type τ, and we define a specific augmentation $\mathcal{O}^*$ called the *full tracker* of $\mathcal{O}$, which aims to keep track of all possible linearizations of a run R of $\mathcal{O}$, as the run unfolds. Recall that by definition, a linearization R_{atom} of a run R is a run of the atomic implementation $\mathcal{O}_{atom}$ of type τ whose behavior matches that of R. In practice, $\overline{\mathcal{O}}$ will actually maintain just the final configuration C_{atom} of each linearization R_{atom}, rather than the whole linearization, thus we now take a closer look at these atomic configurations.

After any run R_{atom}, the final configuration can be characterized by the state $\sigma \in \tau.\Sigma$ of the atomic object, and the states of each of the processes $\pi \in \Pi$. For each process π, there are three types of possible states:

1. either π is *idle*, meaning π has no currently invoked but un-returned operation, i.e. pc_π points at the single line (line a) of the generator algorithm and $r_\pi = \perp$.

2. or π has invoked an operation $oper_\pi(argu)$ which is *yet-to-linearize*, i.e. pc_π points to line 1 of procedure *oper*'s code and $r_\pi = \perp$.

3. or π *has linearized* its operation $oper_\pi(argu)$ with some response $resp$, i.e. pc_π points to line 2 of procedure *oper*'s code and $r_\pi = resp$.

We capture the states of the processes with a function $f(\pi) = (f(\pi).op, f(\pi).arg, f(\pi).res)$, which maps each process π to a triple, where

1. if π is idle, then $f(\pi) = (\perp, \perp, \perp)$
2. if π has invoked an operation $oper_\pi(argu)$ which is yet-to-linearize, then $f(\pi) = (oper, argu, \perp)$
3. and if π has linearized its operation $oper_\pi(argu)$ with some response $resp$, then $f(\pi).op = operation$, $f(\pi) = (oper, argu, resp)$

Thus, we identify each atomic configuration C_{atom}, i.e. configuration of $\mathcal{A}(\mathcal{O}_{atom})$, as an object-state, process-states pair $(C_{atom}.\sigma, C_{atom}.f)$ as described above. Here on, we will refer to these pairs as the atomic configurations.

For the full tracker, we augment the implementation $\mathcal{O}$ with an object $\mathcal{M}^*$ that stores the set of all final configurations of linearizations of the implemented object's run thus far. We call $\mathcal{M}^*$ the *meta-configuration* of the tracker, since it "contains all the atomic configurations in which a corresponding atomic run can be in". The meta-configuration set is, in fact, the only auxiliary object we use in the tracker augmentation, thus $\Omega_{aux} = \{\mathcal{M}^*\}$.

Next, we complete the description of $\mathcal{O}^*$ by describing how the meta-configuration $\mathcal{M}^*$ is initialized and manipulated by the full tracker. Initially, all processes are idle, and $\mathcal{O}$ is in its initial state σ_0; thus we initialize $\mathcal{M}^* = \{(\sigma_0, f_0)\}$, where $f_0(\pi) = (\perp, \perp, \perp)$ for all process $\pi \in \Pi$.

There are three types of lines in a procedure $\mathcal{O}.op_\pi(arg)$: *invocation*, *return*, and *intermediate lines*. The augmenting code at each line ℓ updates $\mathcal{M}^*$, based on ℓ's type. Specifically,

- Case: ℓ is the invocation of $op_\pi(arg)$. We will maintain the invariant that $\mathcal{M}^*$ stores all final configurations of linearizations of the implemented object's current run. Thus, every configuration $C \in \mathcal{M}^*$ will reflect that π is idle before the invocation. In the augmentation to the invocation line, we will update each $C \in \mathcal{M}^*$ to a C' which reflects that π invokes a pending operation *op* with argument arg, and then further evolve C' to reflect that any arbitrary subset of processes with pending operations (possibly including π) can linearize after π invokes its operation. Before formally stating the augmenting code to the invocation line, we develop some helpful notation.

– For an atomic configuration $C = (C.\sigma, C.f)$ in which π is idle, we define:

$$invoke(C, \pi, op_\pi(arg)) \triangleq C'$$

where $C'.\sigma = C.\sigma$, $C'.f(\pi) = (op, arg, \bot)$, and for each $\bar{\pi} \neq \pi$, $C'.f(\bar{\pi}) = C.f(\bar{\pi})$.

– For an atomic configuration C, and a process $\pi \in \Pi$, we define the predicate

$$pending(\pi, C) \equiv C.f(\pi).op \neq \bot \wedge C.f(\pi).res = \bot$$

to capture whether π has a pending operation in C. We further define

$$pending(C) \equiv \{\pi \in \Pi \mid pending(\pi, C.f)\}$$

to be the set of all processes with pending operations in C.

– For any $S \subseteq pending(C)$ of processes with pending operations, and a permutation of those processes $\alpha = (\alpha_1, \ldots, \alpha_{|S|}) \in Perm(S)$, we define the transition function $\delta^*(C, \alpha)$ recursively below. (Informally, $\delta^*(C, (\alpha_1, \ldots, \alpha_k)) = C'$ if applying the pending operations of processes $\alpha_1, \ldots, \alpha_k$, in that order, updates the configuration C to C'.)

- $\delta^*(C, ()) = C$, where $()$ is the empty sequence
- $\delta^*(C, (\alpha_1, \ldots, \alpha_k)) = \delta^*(C', (\alpha_2, \ldots, \alpha_k))$, where

$$\begin{aligned} \exists r \in RES: \quad & (C'.\sigma, r) = \delta(C.\sigma, \alpha_1, C.f(\alpha_1).op, C.f(\alpha_1).arg) \\ & \wedge\ \forall \pi \neq \alpha_1, C'.f(\pi) = C.f(\pi) \\ & \wedge\ C'.f(\alpha_1) = (C.f(\alpha_1).op, C.f(\alpha_1).arg, r) \end{aligned}$$

Formally, the augmenting code at line ℓ is: $\mathcal{M}^* \leftarrow EvolveInv(\mathcal{M}^*, op_\pi(arg))$, where $EvolveInv(\mathcal{M})$ is defined by

$$EvolveInv(\mathcal{M}) \triangleq \left\{ C'' \;\middle|\; \begin{array}{l} \exists C \in \mathcal{M}, C': \\ \quad C' = invoke(C, \pi, op_\pi(arg)) \wedge \\ \quad \exists S \subseteq pending(C'), \alpha \in Perm(S): C'' = \delta^*(C', \alpha) \end{array} \right\} \quad (11.4.1)$$

We say *configuration C'' results from π invoking $op_\pi(arg)$ and $\alpha_1, \ldots, \alpha_k$ linearizing after*

configuration C.

- Case: ℓ is an intermediate line of $op_\pi(arg)$. The augmenting code for this line reflects that any time any process executes an intermediate line, it presents an opportunity for an arbitrary subset of pending operations to linearize in an arbitrary order. Formally, the augmenting code at line ℓ is: $\mathcal{M}^* \leftarrow Evolve(\mathcal{M}^*)$, where $Evolve$ is defined by

$$Evolve(M) \triangleq \{C' \mid \exists C \in M, S \subseteq pending(C), \alpha \in Perm(S) : C' = \delta^*(C, \alpha)\} \qquad (11.4.2)$$

  We say *configuration C' results from $\alpha_1, \ldots, \alpha_k$ linearizing after configuration C.*

- Case: ℓ is a **return** res statement from a procedure $op_\pi(arg)$. Those atomic configurations that do not reflect that the operation has linearized with a response of res are no longer tenable, and thus these are filtered out of $\mathcal{M}^*$. On the other hand, the configurations that show this response of res are retained and updated to reflect that π becomes idle after the return. Furthermore, any arbitrary subset of pending processes can linearize after π returns. Before formally stating the augmenting code to the invocation line, we develop some helpful notation.

  - For an atomic configuration $C = (C.\sigma, C.f)$ in which π has linearized its operation $op_\pi(arg)$ with return value res, we define:

$$return(C, \pi, res) \triangleq C'$$

    where $C'.\sigma = C.\sigma$, $C'.f(\pi) = (\bot, \bot, \bot)$, and for each $\bar{\pi} \neq \pi$, $C'.f(\bar{\pi}) = C.f(\bar{\pi})$.

  Formally, the augmenting code at line ℓ is: $\mathcal{M}^* \leftarrow EvolveRet(\mathcal{M}^*, \textbf{return } res)$, where $EvolveRet$ is defined by

$$EvolveRet(\mathcal{M}) \leftarrow \left\{ C'' \;\middle|\; \begin{array}{l} \exists C \in \mathcal{M}, C' : \\ \quad C.f(\pi).ret = res \wedge \\ \quad C' = return(C, \pi, \textbf{return } res) \wedge \\ \quad \exists S \subseteq pending(C'), \alpha \in Perm(S) : C'' = \delta^*(C', \alpha) \end{array} \right\} \qquad (11.4.3)$$

  We say *configuration C'' results from π returning res and $\alpha_1, \ldots, \alpha_k$ linearizing after config-*

uration C.

Definition 11.4.3 (full tracker)**.** For any implementation $\mathcal{O}$, $\mathcal{O}^*$ is the *full tracker* as specified above with the single auxiliary *meta-configuration* variable $\mathcal{M}^*$.

11.4.3 Main Theorem

The crafting of the tracking augmentation yields a powerful technique for producing *machine verified proofs* of linearizability. This technique falls out from the main theorem of this section, which reduces the complex question of whether an implementation $\mathcal{O}$ is linearizable to the verification of a simple invariant. In particular, $\mathcal{O}$ is linearizable *if and only if* $\mathcal{M}^* \neq \varnothing$ is an invariant of $\mathcal{A}(\mathcal{O}^*)$.

Exploiting the *if* direction of this theorem, we can obtain machine verified proofs of linearizability by proving the invariant in a software system such as TLAPS (Temporal Logic of Actions Proof System) or Coq. The *only if* direction of the theorem assures us that this proof technique is complete.

Theorem 11.4.4. *Let $\mathcal{O}$ be an implementation of an object of type τ initialized to state σ_0 for a set of processes* Π*, $\mathcal{O}$ is linearizable* if and only if *$\mathcal{M}^* \neq \varnothing$ is an invariant of $\mathcal{A}(\mathcal{O}^*)$.*

(Recall that $\mathcal{O}^*$ is the full tracker of $\mathcal{O}$ with meta-configuration $\mathcal{M}^*$, and $\mathcal{A}(\mathcal{O}^*)$ is the algorithm from Figure 11.3.1 in which processes repeatedly make calls to procedures of $\mathcal{O}^*$.)

The remainder of the section develops the necessary machinery and proves Theorem 11.4.4.

Let $\mathcal{O}$ be an implementation of an object of type τ initialized to σ_0 for a set of processes Π, and let O be an atomic object with the same type and initialization, and for the same set of processes. Finally, recall that $\mathcal{O}^*$ is the full tracker of $\mathcal{O}$ with meta-configuration variable $\mathcal{M}^*$.

For a generic variable V of any given implementation I, and a run R_I of the generator $\mathcal{A}(I)$, we let $V(R_I)$ denote the value of V in the final configuration of $\mathcal{A}(I)$. We will be particularly interested in the meta-configurations $\mathcal{M}^*(R^*)$ for runs R^* of $\mathcal{A}(\mathcal{O}^*)$.

By Observation 2, $\mathcal{O}$ is linearizable if and only if the finite behaviors of $\mathcal{O}^*$ are behaviors of O. Our strategy to prove the main theorem thereby, will be to show that for any run R^* of $\mathcal{O}^*$, the set of atomic runs sharing its behavior will have exactly the final configurations $\mathcal{M}^*(R^*)$.

To effectively express the above statement, we develop a bit more notation. For any behavior B, we let $\mathit{AtomicRuns}(B)$ be the set of atomic runs that exhibit behavior B. Further, for run R, we let $\mathcal{C}(R)$ denote the final configuration of R, and for a set of runs $\mathcal{R}$, we let $\mathcal{C}(\mathcal{R}) = \{\mathcal{C}(R) \mid R \in \mathcal{R}\}$ denote the set of final configurations of all those runs. For a given run R^* of the implementation, we

define $Linearizations(R^*) \triangleq AtomicRuns(behavior(R^*))$ to be the set of all possible linearizations of R^*.

We are now ready to prove the key lemma that will propel the proof of Theorem 11.4.4.

Lemma 11.4.5. *Let R^* be any finite run of $\mathcal{O}^*$, then the following equality holds:*

$$\mathcal{M}^*(R^*) = \mathcal{C}(Linearizations(R^*))$$

Proof. The proof is by induction on the number of events in the run R^*.

- **Base Case:** Every implementation run R^* with zero events is simply an initial configuration C_0^* of algorithm $\mathcal{A}(\mathcal{O}^*)$. In this case, $behavior(R^*)$ is the empty behavior, and thus only the empty atomic run is in $Linearizations(R^*) = AtomicRuns(behavior(R^*))$. The final (and only) configuration of this run is the unique initial atomic configuration $C_0 = (\sigma_0, f_0)$; thus $\mathcal{C}(Linearizations(R^*)) = \{(\sigma_0, f_0)\}$ which is the definition of $\mathcal{M}^*(C_0^*)$.

- **Induction Step:** Let $R_k^* = C_0^*, \ldots, C_{k-1}^*, (\pi_k, \ell_k), C_k^*$ be run with $k \geq 1$ events. We consider the prefix-run $R_{k-1}^* = C_0^*, \ldots, C_{k-1}^*$, and note that the induction hypothesis states that:

  $$\mathcal{M}^*(R_{k-1}^*) = \mathcal{C}(Linearizations(R_{k-1}^*))$$

  For notational convenience, we define $B_k = behavior(R_k^*)$ and $B_{k-1} = behavior(R_{k-1}^*)$. The last line executed in R^*, i.e. ℓ_k, can be one of three types of lines: invocation, intermediate, or return.

  - Case: ℓ_k is an invocation $op_\pi(arg)$. First, we show that RHS $\subseteq$ LHS. That is, let $L_k \in Linearizations(R_k^*)$; we must first prove that $\mathcal{C}(L_k) \in \mathcal{M}^*(R_k^*)$. To this effect, define L_{k-1} to be the longest prefix of L_k, such that $behavior(L_{k-1}) = B_{k-1}$. By definition of atomic runs, L_k results when L_{k-1} is followed immediately by the invocation $op_\pi(arg)$ and subsequently by a (possibly empty) sequence of linearization steps by processes $\alpha = \alpha_1, \ldots, \alpha_h$ that are yet to linearize in $invoke(\mathcal{C}(L_{k-1}), \pi, op_\pi(arg))$. In particular, no other invocation or return events are possible, since $behavior(L_k) = B_k$ which has just the single invocation more than B_{k-1}. Thus, by definition $\mathcal{C}(L_k)$ results from π invoking $op_\pi(arg)$ and $\alpha_1, \ldots, \alpha_h$ linearizing after configuration $\mathcal{C}(L_{k-1})$. Since $\mathcal{C}(L_{k-1}) \in \mathcal{M}^*(R_{k-1}^*)$ by the inductive hypothesis, $\mathcal{C}(L_k) \in EvolveInv(\mathcal{M}^*(R_{k-1}^*), \pi, op_\pi(arg)) =$

$\mathcal{M}^*(R_k^*)$.

Second we prove that LHS $\subseteq$ RHS. That is, let $C_k \in \mathcal{M}^*(R_k^*)$, we must prove that there is a linearization L_k of R_k^* with final configuration $\mathcal{C}(L_k) = C_k$. To this effect, we note that $\mathcal{M}^*(R_k^*) = EvolveInv(\mathcal{M}^*(R_{k-1}^*), \pi, op_\pi(arg))$. That is, there is a configuration $C_{k-1} \in \mathcal{M}^*(R_{k-1}^*)$ such that C_k results from π invoking $op_\pi(arg)$ and some sequence of processes $\alpha = \alpha_1, \ldots, \alpha_h$ linearizing after C_{k-1}. By the inductive hypothesis, there is a linearization L_{k-1} of R_{k-1}^* whose final configuration is $\mathcal{C}(L_{k-1}) = C_{k-1}$ in which π is idle and $\alpha_1, \ldots, \alpha_h$ are yet-to-linearize in $invoke(C_{k-1}, \pi, op_\pi(arg))$. Thus, the atomic run L_k that extends L_{k-1} with π invoking and α linearizing after C_{k-1} is a linearization of R_k^* whose final configuration is $\mathcal{C}(L_k) = C_k$, which concludes the proof of the case.

– Case: ℓ_k is an intermediate line. Since $behavior(R_k^*) = behavior(R_{k-1}^*)$, we invoke the inductive hypothesis to get the equality:

$$\mathcal{M}^*(R_{k-1}^*) = \mathcal{C}(Linearizations(R_{k-1}^*)) = \mathcal{C}(Linearizations(R_k^*))$$

Now, we observe that when $\mathcal{M}^*(R_{k-1}^*)$ is already the full set of final configurations of linearizations with the behavior $B_{k-1} = B_k$, evolving does not change the set, that is:

$$\mathcal{M}^*(R_k^*) = Evolve(\mathcal{M}^*(R_{k-1}^*)) = Evolve(\mathcal{C}(Linearizations(R_k^*))) = \mathcal{C}(Linearizations(R_k^*))$$

Thus, we conclude the proof of the case.

– Case: ℓ_k is a return **return** res. This case is very similar to the first case. Once again, we first show that RHS $\subseteq$ LHS. That is, let $L_k \in Linearizations(R_k^*)$; we must first prove that $\mathcal{C}(L_k) \in \mathcal{M}^*(R_k^*)$. To this effect, define L_{k-1} to be the longest prefix of L_k, such that $behavior(L_{k-1}) = B_{k-1}$. By definition of atomic runs, L_k results when L_{k-1} is followed immediately by the return of res and subsequently by a (possibly empty) sequence of linearization steps by processes $\alpha = \alpha_1, \ldots, \alpha_h$ that are yet to linearize in $return(\mathcal{C}(L_{k-1}), \pi, \mathbf{return}\, res)$. In particular, no other invocation or return events are possible, since $behavior(L_k) = B_k$ which has just the single return more than B_{k-1}. Thus, by definition $\mathcal{C}(L_k)$ results from π returning res and $\alpha_1, \ldots, \alpha_h$ linearizing after configuration $\mathcal{C}(L_{k-1})$. Since $\mathcal{C}(L_{k-1}) \in \mathcal{M}^*(R_{k-1}^*)$ by the inductive hypothesis, $\mathcal{C}(L_k) \in EvolveRet(\mathcal{M}^*(R_{k-1}^*), \pi, \mathbf{return}\, res) = \mathcal{M}^*(R^*)$.

Second we prove that LHS $\subseteq$ RHS. That is, let $C_k \in \mathcal{M}^*(R_k^*)$, we must prove that there is a linearization L_k of R_k^* with final configuration $\mathcal{C}(L_k) = C_k$. To this effect, we note that $\mathcal{M}^*(R_k^*) = \mathit{EvolveRet}(\mathcal{M}^*(R_{k-1}^*), \pi, \textbf{return}\ res)$. That is, there is a configuration $C_{k-1} \in \mathcal{M}^*(R_{k-1}^*)$ such that C_k results from π returning res and some sequence of processes $\alpha = \alpha_1, \ldots, \alpha_h$ linearizing after C_{k-1}. By the inductive hypothesis, there is a linearization L_{k-1} of R_{k-1}^* whose final configuration is $\mathcal{C}(L_{k-1}) = C_{k-1}$ in which π has linearized with return value res and $\alpha_1, \ldots, \alpha_h$ are yet-to-linearize in $\mathit{return}(C_{k-1}, \pi, \textbf{return}\ res)$. Thus, the atomic run L_k that extends L_{k-1} with π returning and α linearizing after C_{k-1} is a linearization of R_k^* whose final configuration is $\mathcal{C}(L_k) = C_k$, which concludes the proof of the case.

□

We are now ready to prove the main theorem.

Theorem 11.4.4. *Let $\mathcal{O}$ be an implementation of an object of type τ initialized to state σ_0 for a set of processes* Π*, $\mathcal{O}$ is linearizable* if and only if $\mathcal{M}^* \neq \varnothing$ *is an invariant of $\mathcal{A}(\mathcal{O}^*)$.*

(Recall that $\mathcal{O}^*$ is the full tracker of $\mathcal{O}$ with meta-configuration $\mathcal{M}^*$, and $\mathcal{A}(\mathcal{O}^*)$ is the algorithm from Figure 11.3.1 in which processes repeatedly make calls to procedures of $\mathcal{O}^*$.)

Proof. We will prove the theorem by showing the forward and reverse directions separately:

- For the forward direction: assume $\mathcal{M}^* \neq \varnothing$ is an invariant of $\mathcal{A}(\mathcal{O}^*)$. Thus, for any finite run R^* of $\mathcal{A}(\mathcal{O}^*)$, we note $\mathcal{M}^*(R^*)$ is non-empty. So, by Lemma 11.4.5, there is a linearization L of R^*. Since, an arbitrary finite run R^* of $\mathcal{A}(\mathcal{O}^*)$ is linearizable, we conclude that $\mathcal{O}^*$ is linearizable. Since $\mathcal{O}^*$ has the same behaviors as $\mathcal{O}$, thus $\mathcal{O}$ is linearizable.

- For the reverse direction: assume that $\mathcal{O}$ is linearizable. This implies $\mathcal{O}^*$ is linearizable. So, every finite run R^* of $\mathcal{A}(\mathcal{O}^*)$ has at least one linearization L. Thus, by Lemma 11.4.5, $\mathcal{M}^*(R^*)$ is non-empty. Since, $\mathcal{M}^*(R^*)$ is non-empty for an arbitrary finite run R^* of $\mathcal{A}(\mathcal{O}^*)$, we conclude that $\mathcal{M}^* \neq \varnothing$ is an invariant of $\mathcal{A}(\mathcal{O}^*)$.

□

11.5 (Partial) Trackers

The proof of Theorem 11.4.4 demonstrates that, for an implementation $\mathcal{O}$, the full tracker $\mathcal{O}^*$, in essence, tracks "all possible linearizations" in its meta-configuration. To show that $\mathcal{O}$ is linearizable however, it suffices to show the existence of a single linearization. That is, we need not track *all* linearizations, but just ensure that we track *at least one*. In practice, tracking a subset of linearizations can be easier for a prover, especially one who knows the structure of the implementation well. Thus, in general, we will be interested in a (partial) tracker, whose meta-configuration $\mathcal{M}'$ maintains a subset of the full meta-configuration $\mathcal{M}^*$. We define such trackers.

Definition 11.5.1 (trackers)**.** Let $\mathcal{O}$ be an implementation of a type τ in the initial state σ_0 for a set of processes Π. We say an augmentation $\mathcal{O}'$ is a *(partial) tracker* of $\mathcal{O}$, if the following conditions are met:

- The auxiliary variable set Ω_{aux} of $\mathcal{O}'$ contains a meta-configuration variable $\mathcal{M}'$ that holds a set of atomic configurations. (Ω_{aux} may also contain other additional variables.)
- $\mathcal{M}'$ is initialized to $\{(\sigma_0, f_0)\}$, where $f_0(\pi) = (\bot, \bot, \bot)$ for every $\pi \in \Pi$.
- For each type of line ℓ (whether invocation, intermediate, or return) in the original algorithm, the augmented line ℓ' updates the tracker to ensure the new value of $\mathcal{M}'$ is a subset of the configurations that would arise by evolving $\mathcal{M}'$ according to the rules of the full tracker. That is, if the update rule of the tracker at line ℓ' is $\mathcal{M}' \leftarrow \mathit{Evolve}_\ell(\mathcal{M}')$, then:
  - Case: ℓ is the invocation of $op_\pi(arg)$.

$$\mathit{Evolve}_\ell(\mathcal{M}') \subseteq \mathit{EvolveInv}(\mathcal{M}', \pi, op_\pi(arg)) \tag{11.5.1}$$

  - Case: ℓ is an intermediate line of $op_\pi(arg)$.

$$\mathit{Evolve}_\ell(\mathcal{M}') \subseteq \mathit{Evolve}(\mathcal{M}') \tag{11.5.2}$$

  - Case: ℓ is a **return** res statement from a procedure $op_\pi(arg)$.

$$\mathit{Evolve}_\ell(\mathcal{M}') \subseteq \mathit{EvolveRet}(\mathcal{M}', \pi, \textbf{return}\ res) \tag{11.5.3}$$

Theorem 11.5.2. *Let $\mathcal{O}$ be an implementation of an object of type τ initialized to state σ_0 for a set of processes Π, and $\mathcal{O}'$ be any partial tracking augmentation of $\mathcal{O}$ with meta-configuration variable $\mathcal{M}'$. If $\mathcal{M}' \neq \varnothing$ is an invariant of $\mathcal{A}(\mathcal{O}')$, then $\mathcal{O}$ is linearizable.*

Proof. Let $\mathcal{M}^*$ be the meta-configuration of the full tracker $\mathcal{O}^*$. Consider any run $R^* = C_0^*, (\pi_1, \ell_1^*), C_1^*, (\pi_2, \ell_2^*), \ldots$ of $\mathcal{A}(\mathcal{O}^*)$, and recall that we say the run $R' = C_0', (\pi_1, \ell_1'), C_1', (\pi_2, \ell_2'), \ldots$ of $\mathcal{A}(\mathcal{O}')$ is coupled with R^*, if C_0' has the same initialization for the private registers and objects in $\tau.\Omega$, and for each i, ℓ_i' is the line in $\mathcal{O}'$ that corresponds to ℓ_i^* in $\mathcal{O}^*$, i.e., these are augmentations of corresponding lines in $\mathcal{O}$. Since $\mathcal{O}^*$ and $\mathcal{O}'$ augment the same implementation, all runs of $\mathcal{A}(\mathcal{O}^*)$ and $\mathcal{A}(\mathcal{O}')$ are coupled as (R^*, R') in this way. For coupled runs R^* and R', by definition of a partial tracking augmentation, it is clear that $\mathcal{M}' \subseteq \mathcal{M}^*$ throughout the coupled run. Thus, $\mathcal{M}' \neq \varnothing$ being an invariant of $\mathcal{O}'$ implies $\mathcal{M}^* \neq \varnothing$ is an invariant of $\mathcal{O}^*$. So, by Theorem 11.4.4, the proof is complete. □

11.6 Proving Strong Linearizability

In the previous section, we motivated partial trackers as simpler ways to prove standard linearizability. In this section, we will show how to use partial trackers to obtain a *sound and complete* method for proving *strong linearizability*.

The pivotal difference between strong linearizability and (standard) linearizability, is that the former consistency condition requires that we should be able to commit to a single linearization $\mathcal{L}(R)$ for any run R, such that the linearization of every extension of R will be an extension of $\mathcal{L}(R)$. Our key insight lies here. The full tracker's meta-configuration $\mathcal{M}^*$ maintains the final configurations of *every possible linearization* of R hoping to extend whichever ones may work as the future of the run unfolds; thus, it effectively ensures that "no linearization will be missed" thereby yielding a complete method for proving linearizability. However, informally speaking, by maintaining every possible linearization, it does exactly the opposite of *committing to single linearization*, which is the key to showing strong linearizability.

Our idea therefore, is to demonstrate strong linearizability by demonstrating a partial tracker $\mathcal{O}'$ whose meta-configuration $\mathcal{M}'$ holds precisely one configuration at any point in time. The intuition is that maintaining a single configuration $\mathcal{M}'(R_k') = \{C_k\}$ at the end of a run R_k' of length k is akin to committing to a single linearization L_k (with $\mathcal{C}(L_k) = C_k$) for this run, and that this unique linearization is getting extended as the run gets extended. Of course, there is a small catch. The configuration C_k could be the final configuration of many different linearizations, which all happen

to have the same final configuration. Nevertheless, we will resolve this hiccup, by proving that a cleverly chosen particular one of these linearizations can be picked as $\mathcal{L}(R'_k)$.

Theorem 11.6.1. *Let $\mathcal{O}$ be an implementation of an object of type τ initialized to state σ_0 for a set of processes Π, $\mathcal{O}$ is strongly linearizable* if and only if *there exists a partial tracker $\mathcal{O}'$ such that $\mathcal{A}(\mathcal{O}')$ satisfies the invariant $|\mathcal{M}'| = 1$.*

(Recall that $\mathcal{O}'$ is the partial tracker of $\mathcal{O}$ with meta-configuration $\mathcal{M}'$, and $\mathcal{A}(\mathcal{O}')$ is the algorithm from Figure 11.3.1 in which processes repeatedly make calls to procedures of $\mathcal{O}'$.)

Proof. We split the proof into two parts, proving the *only if* direction, and then proving the *if* direction.

1. To prove the *only if* direction, we assume there is a linearization function $\mathcal{L}$ that maps each finite run $R_k = C_0, (\pi_1, \ell_1), C_1, \ldots, (\pi_k, \ell_k), C_k$ of $\mathcal{A}(\mathcal{O})$ to a linearization $\mathcal{L}(R_k)$. We claim there is a tracker $\mathcal{O}'$ which maintains a meta-configuration $\mathcal{M}'$, whose value $\mathcal{M}'(R'_k)$ after any run R'_k that is coupled with run R_k of $\mathcal{O}$ is equal to $\mathcal{M}'(R'_k) = \{\mathcal{L}(R_k)\}$. We prove this claim by induction.

   Base Case: note that if $R_k = R_0$ is a zero-event run, then its only linearization is the zero-event atomic run $L_0 = C_0$ that (starts and) ends in the initial atomic configuration (σ_0, f_0), thus $\mathcal{L}(R_0) = L_0$ and $\mathcal{M}'$'s initialization is indeed $\{(\sigma_0, f_0)\}$, which concludes the base case.

   Induction Step: if $\mathcal{M}'(R'_k) = \{\mathcal{L}(R_k)\}$ for some $k \geq 0$, and the run R_k extends to R_{k+1} in a single step, then by assumption $L_{k+1} = \mathcal{L}(R_{k+1})$ is an extension of $L_k = \mathcal{L}(R_k)$. Here, we break the argument into cases, depending on which type of line ℓ_{k+1} is (invocation, intermediate, or return):

   (a) Case: ℓ is the invocation of $op_\pi(arg)$. $\mathcal{C}(L_{k+1})$ must result from some processes $\alpha_1, \ldots, \alpha_{h_1}$ linearizing, then π_{k+1} invoking $op_\pi(arg)$ then $\alpha_{h_1+1}, \ldots, \alpha_{h_2}$ linearizing after $\mathcal{C}(L_k)$ for some $0 \leq h_1 \leq h_2$. However, since linearizing an operation before or after an invocation makes no difference to the final configuration, we can obtain the same $\mathcal{C}(L_{k+1})$ after $\mathcal{C}(L_k)$ by π invoking $op_\pi(arg)$ and then linearizing $\alpha_1, \ldots, \alpha_{h_2}$. Thus, $\mathcal{C}(L_{k+1}) \in EvolveInv(\mathcal{M}'(R'_k), \pi, op_\pi(arg))$.

   (b) Case: ℓ is an intermediate line of $op_\pi(arg)$. $\mathcal{C}(L_{k+1})$ must result from linearizing some sequence of processes $\alpha_1, \ldots, \alpha_h$ after $\mathcal{C}(L_k)$. Thus, $\mathcal{C}(L_{k+1}) \in Evolve(\mathcal{M}'(R'_k))$.

(c) Case: ℓ is a **return** res statement from a procedure $op_\pi(arg)$. $\mathcal{C}(L_{k+1})$ must result from some processes $\alpha_1, \ldots, \alpha_{h_1}$ linearizing, then π_{k+1} returning res then $\alpha_{h_1+1}, \ldots, \alpha_{h_2}$ linearizing after $\mathcal{C}(L_k)$ for some $0 \leq h_1 \leq h_2$. However, since linearizing an operation before or after a return makes no difference to the final configuration, we can obtain the same $\mathcal{C}(L_{k+1})$ after $\mathcal{C}(L_k)$ by π returning res and then linearizing $\alpha_1, \ldots, \alpha_{h_2}$. Thus, $\mathcal{C}(L_{k+1}) \in \mathit{EvolveRet}(\mathcal{M}'(R'_k), \pi, \textbf{return}\, res)$.

That completes the proof of the *only if* direction.

2. To prove the *if* direction, we assume that there is a tracker $\mathcal{O}'$ that always maintains a singleton meta-configuration $\mathcal{M}'$. We now recursively define a prefix preserving linearization function $\mathcal{L}$ on runs of $\mathcal{A}(\mathcal{O})$, such that for coupled runs R of $\mathcal{O}$ and R' of $\mathcal{O}'$, $\mathcal{M}'(R') = \mathcal{C}(\mathcal{L}(R))$:

   - If R_0 is a zero event run of $\mathcal{A}(\mathcal{O})$, define $\mathcal{L}(R_0) \triangleq (\sigma_0, f_0)$. (This is clearly a (indeed the only) linearization of R_0.)
   - If R_k is a k event run for $k \geq 0$, consider the prefix run R_{k-1} such that $R_k = R_{k-1}, (\pi_k, \ell_k), \mathcal{C}(R_k)$. Also consider the corresponding $k-1$ and k event coupled runs of $\mathcal{O}'$: R'_{k-1} and R'_k, and their meta-configurations: $\mathcal{M}'(R'_{k-1}) = \{C_{k-1}\}$ and $\mathcal{M}'(R'_k) = \{C_k\}$. We now finish the definition with three cases:
     (a) Case: ℓ is the invocation of $op_\pi(arg)$. C_k must result from π invoking $op_\pi(arg)$ and some sequence of processes $\alpha_1, \ldots, \alpha_h$ linearizing after $C_{k-1} = \mathcal{L}(R_{k-1})$. We define $\mathcal{L}(R_k)$ to be the run resultant from $\mathcal{L}(R_{k-1})$ being extended by that invocation and those linearization events.
     (b) Case: ℓ is an intermediate line of $op_\pi(arg)$. C_k must result from some sequence of processes $\alpha_1, \ldots, \alpha_h$ linearizing after $C_{k-1} = \mathcal{L}(R_{k-1})$. We define $\mathcal{L}(R_k)$ to be the run resultant from $\mathcal{L}(R_{k-1})$ being extended by those linearization events.
     (c) Case: ℓ is a **return** res statement from a procedure $op_\pi(arg)$. C_k must result from π returning res and some sequence of processes $\alpha_1, \ldots, \alpha_h$ linearizing after $C_{k-1} = \mathcal{L}(R_{k-1})$. We define $\mathcal{L}(R_k)$ to be the run resultant from $\mathcal{L}(R_{k-1})$ being extended by that return and those linearization events.

   By construction, we see that $\mathcal{L}$ is a prefix preserving linearization function of the runs of implementation $\mathcal{O}$. That concludes the proof.

□

11.7 Example: The Union Find Object

In this section, we consider Jayanti and Tarjan's concurrent union-find implementation [118, 119], and describe how we wrote the machine verified proof that it is linearizable, in fact strongly linearizable, using TLAPS (the Temporal Logic of Actions Proof System) [153]. We chose Jayanti and Tarjan's algorithm since it is simple (both to understand and in length of code) and highly practical—it is the fastest algorithm for computing connected components of a graph on CPUs [51] and GPUs [95], and has several other applications [3].

11.7.1 The Union Find Type

The union find type maintains a partition of the *elements* in $[n] = \{1, 2, \ldots, n\}$ and supports two operations.

- $\textsc{Find}(x)$ returns the maximum element in element x's part of the partition.
- $\textsc{Unite}(x, y)$ merges the parts containing x and y if they are different and returns *ack*.

Formally, we specify the partition as a function $\sigma : [n] \to \mathcal{P}([n])$ from the set of elements to the powerset of the set of elements, such that $\sigma(x)$ is the part containing element x. Of course, if two elements x and y are in the same part of the partition, then $\sigma(x) = \sigma(y)$. We give the full formal specification of the union-find type in the figure Object Type 11.7.1.

Object Type 11.7.1 (Union Find Object)**.** A union find type of n elements $[n] = \{1, \ldots, n\}$ is described as follows:

- $\Sigma = \left\{ \sigma : [n] \to \mathcal{P}([n]) \;\middle|\; \{\sigma(x) \mid x \in [n]\} \text{ is a partition of } [n], \text{ and } \forall x \in [n] : x \in \sigma(x) \right\}$
- $OP = \{\textsc{Unite}, \textsc{Find}\}$
- $ARG_{\textsc{Unite}} = [n] \times [n], ARG_{\textsc{Find}} = [n]$
- $RES = \{ack\} \cup [n]$
- Transition function δ is defined by:
  - $\delta(\sigma, \pi, \textsc{Find}, x) = \max \sigma(x)$
  - $\delta(\sigma, \pi, \textsc{Unite}, (x, y)) = \begin{cases} (\sigma, ack), & \text{if } \sigma(x) = \sigma(y) \\ (\sigma', ack), & \text{if } \sigma(x) \neq \sigma(y) \\ & \text{where } \forall z \notin \sigma(x) \cup \sigma(y), \sigma'(z) = \sigma(z) \\ & \text{and} \quad \forall z \in \sigma(x) \cup \sigma(y), \sigma'(z) = \sigma(x) \cup \sigma(y) \end{cases}$

11.7.2 The Jayanti Tarjan Union Find Implementation

We present Jayanti and Tarjan's implementation of union-find in Figure 11.7.1. Each numbered line in the implementation requires the performance of at most one shared memory instruction, and is performed atomically.

Base Objects: $x.par$ is a Read/CAS register initialized to $x.par = x$, for each node $x \in [n]$.

1: **procedure** $\mathcal{O}.\text{FIND}_\pi(x_\pi)$

  $u_\pi \leftarrow x_\pi$

2: **while** $u_\pi \neq u_\pi.par$ **do** $u_\pi \leftarrow u_\pi.par$

3: **return** u_π

4: **procedure** $\mathcal{O}.\text{UNITE}_\pi(x_\pi, y_\pi)$

  $u_\pi \leftarrow x_\pi$; $v_\pi \leftarrow y_\pi$

  while *true* **do**

5: **if** $u_\pi = v_\pi$ **then goto** line 8

  else if $u_\pi < v_\pi$ **then if** $\text{CAS}(u_\pi.par, u_\pi, v_\pi)$ **then goto** line 8

  else if $u_\pi > v_\pi$ **then if** $\text{CAS}(v_\pi.par, v_\pi, u_\pi)$ **then goto** line 8

6: **while** $u_\pi \neq u_\pi.par$ **do** $u_\pi \leftarrow u_\pi.par$

7: **while** $v_\pi \neq v_\pi.par$ **do** $v_\pi \leftarrow v_\pi.par$

8: **return** *ack*

Figure 11.7.1: : Jayanti and Tarjan's implementation of a union find object on n nodes $\{1, 2, \ldots, n\}$ each initially in its own singleton part of the partition, i.e., the initial state is $\sigma_0 : [n] \to \mathcal{P}([n])$ defined by $\forall x \in [n], \sigma_0(x) = x$.

There are two goals to this section: (1) *to convince* the reader (our current verifier) that the implementation in Figure 11.7.1 is strongly linearizable; (2) to *describe the process* by which we prove this strong linearizability. Meeting goal (1) is very simple: the reader simply needs to verify that Figure 11.7.2 is indeed a partial tracker of this implementation with meta-configuration variable $\mathcal{M}$, and then to go to:

https://github.com/visveswara/machine-certified-linearizability/blob/master/UnionFindTracker.tla

where we give a machine verified proof that $|\mathcal{M}| = 1$ is an invariant of the partial tracker. Done! In particular, the reader *does not* need to read or understand the proof of the invariant; in fact,

the reader *does not* even need to understand the code of the implementation! It simply suffices to check that the machine has verified the invariant. Thus, the rest of this section is primarily aimed at meeting goal (2), that is, describing the process by which we—the designers of the proof—went about constructing the partial tracker and what work is involved in producing a proof of the invariant $|\mathcal{M}| = 1$ that the TLAPS proof assistant is able to machine-verify. We start by describing the union-find implementation.

In the implementation, each element is represented by a node, and each node x has a parent pointer field $x.par$ that points to another node. Each part of the partition is represented as a single parent pointer tree, so $\sigma(x) = \sigma(x.par)$, and we maintain the invariant that the parent of x is always greater than or equal to x (i.e. $x.par \geq x$). Thus, the roots of the trees are the largest elements in their respective partition; if u is a root of its tree than $u.par = u$. In the initial state σ_0, all elements are in their own singleton part of the partition, i.e. $\forall x \in [n], \sigma_0(x) = \{x\}$. Correspondingly, the implementation starts with $\forall x \in [n], x.par = x$.

With this representation, a process π performs $\textsc{Find}_\pi(x_\pi)$ by starting a node u_π at x_π (line 1), walking u_π up the parent pointers until it reaches a root (line 2), and returning that root, which must be the maximum element in the set (line 3).

A process π performs $\textsc{Unite}_\pi(x_\pi, y_\pi)$ by starting a node u_π at x_π and another node v_π at y_π (line 4). The implementation ambitiously hopes that the current nodes u_π and v_π are roots. If so, and if $u_\pi = v_\pi$ then unite has no work to do and proceeds to return (first part of line 5); otherwise, if $u_\pi < v_\pi$, then unite tries to merge the two separate trees by linking u_π under v_π via a CAS (second part of line 5); in the remaining case that $u_\pi > v_\pi$ it tries to make the link in the other direction (third part of line 5). If a link gets made, then there is no more work to do so π proceeds to return. Otherwise, the links may have failed since u_π or v_π was not a root. In this case, the implementation walks u_π up the tree until it is a root (line 6), and does the same for v_π (line 7) and tries again (while true loop). Once one of the three cases on line 5 eventually occurs, π returns *ack* (line 8).

11.7.3 The Tracker

Having been given the implementation in Figure 11.7.1, we can blindly write down the full tracker for the implementation, and could prove its linearizability by showing the invariance of $\mathcal{M}^* \neq \varnothing$ by appealing to Theorem 11.4.4. However, we will instead go the route of constructing a partial tracker for two reasons: (1) we would like to prove *strong linearizability*, rather than just lineariz-

ability, and (2) the full tracker exhaustively maintains the set of all possible linearizations of every run by maintaining atomic configurations that arise by every possible set of processes linearizing in every order after every step of the run. We however, will demonstrate a more insightful understanding of exactly when and how the implemented object linearizes operations. By embedding this understanding into the partial tracker, we will make our job of proving the invariant easier.

Our key insight into the implementation is our ability to identify a unique linearization point for each operation, as described in the following observation:

Observation 3. For a FIND, the last iteration of the loop at line 2 is when π discovers that u_π is its own parent and thereby the root and maximum value node in its tree; we posit this to be the linearization point of the operation. For a UNITE, the last execution of line 5 is when π either discovers that u_π and v_π are already in the same tree or when π successfully links their trees together with a CAS; so we posit this to be the linearization point.

For now, the linearization points in Observation 3 are simply an expression of our intuition from an informal understanding of the algorithm. However, we will use this intuition to design our partial tracker presented in Figure 11.7.2, and the proof of our tracker's invariants, which we will discuss later, will formalize this understanding.

We ensure that our implementation in Figure 11.7.2 meets the definition of a tracker, by ensuring that the meta-configuration variable $\mathcal{M}$ meets the initialization, invocation, intermediate line, and return line criteria. In particular:

Base Objects:

- $x.par$ is a Read/CAS register initialized to $x.par = x$, for each node $x \in [n]$.
- $\mathcal{M}$ initialized to $\{(\sigma_0, f_0)\}$ is a meta-configuration, where σ_0 maps each $x \in [n]$ to $\{x\}$ and f_0 maps each $\pi \in \Pi$ to $(\perp, \perp, \perp)$.

1: **procedure** $\overline{\mathcal{O}}.\textsc{Find}_\pi(x_\pi)$

$u_\pi \leftarrow x_\pi$

$\mathcal{M} \leftarrow \{C' \mid \exists C \in \mathcal{M} : C' = invoke(C, \pi, \textsc{Find}_\pi(x_\pi))\}$

2: **while** $u_\pi \neq u_\pi.par$ **do** $u_\pi \leftarrow u_\pi.par$

if $u_\pi = u_\pi.par$ **then** $\mathcal{M} \leftarrow \left\{ C' \middle| \ \exists C \in \mathcal{M} : C' = \delta^*(C, \pi) \ \right\}$

3: **return** u_π

$\mathcal{M} \leftarrow \{C' \mid \exists C \in \mathcal{M} : C' = return(C, \pi, \textbf{return}\ u_\pi)\}$

4: **procedure** $\overline{\mathcal{O}}.\textsc{Unite}_\pi(x_\pi, y_\pi)$

$u_\pi \leftarrow x_\pi$; $v_\pi \leftarrow y_\pi$

$\mathcal{M} \leftarrow \{C' \mid \exists C \in \mathcal{M} : C' = invoke(C, \pi, \textsc{Unite}_\pi(x_\pi, y_\pi))\}$

while *true* **do**

5: **if** $u_\pi = v_\pi$ **then goto** line 8

else if $u_\pi < v_\pi$ **then if** $\textsc{Cas}(u_\pi.par, u_\pi, v_\pi)$ **then goto** line 8

else if $u_\pi > v_\pi$ **then if** $\textsc{Cas}(v_\pi.par, v_\pi, u_\pi)$ **then goto** line 8

if $(u_\pi = v_\pi) \vee (u_\pi < v_\pi \wedge u_\pi = u_\pi.par) \vee (u_\pi > v_\pi \wedge v_\pi = v_\pi.par)$ **then**

$\mathcal{M} \leftarrow \{C' \mid \exists C \in \mathcal{M} : C' = \delta^*(C, \pi)\}$

6: **while** $u_\pi \neq u_\pi.par$ **do** $u_\pi \leftarrow u_\pi.par$

7: **while** $v_\pi \neq v_\pi.par$ **do** $v_\pi \leftarrow v_\pi.par$

8: **return** *ack*

$\mathcal{M} \leftarrow \{C' \mid C' = return(C, \pi, \textbf{return}\ ack)\}$

Figure 11.7.2: : Tracker $\overline{\mathcal{O}}$ for the union find implementation $\mathcal{O}$ presented in Figure 11.7.1.

0. Our meta-configuration is initialized to $\mathcal{M} = \{(\sigma_0, f_0)\}$, where σ_0 is the initial state of the union-find object, and f_0 describes each process as initially idle.

1. In each invocation line by a process π, by the definition of a partial tracker, we can update $\mathcal{M}$

to contain any configurations C' that result from the invocation and an arbitrary sequence of processes that have pending operations linearizing after any $C \in \mathcal{M}$. Observation 3 suggests that we need not think of any operations as linearizing at this line, thus we update $\mathcal{M}$ to contain exactly the subset of configurations which result from the invocation occurring (and the empty sequence of processes linearizing) after some $C \in \mathcal{M}$. This can be confirmed by inspecting the augmentations at line 1 (for FIND) and line 4 (for UNITE).

2. In each intermediate line by a process π, by the definiton of a partial tracker, we can update $\mathcal{M}$ to contain any configurations C' that result from an arbitrary sequence of processes that have pending operations linearizing after any $C \in \mathcal{M}$. Observation 3 suggests that we can consider at most one process linearing after each line of code. In particular, the observation suggests that for FIND, we need to only π linearizing at line 2 if $u_\pi = u_\pi.par$ at the time of that line's execution. Similarly, for UNITE, we need to only consider π linearizing at line 5, and that too only if either $u_\pi = v_\pi$ or if the conditions are such that one of the two CAS operations will succeed on that line. We correspondingly design $\mathcal{M}$ to be updated according to these rules at these particular lines. Otherwise, we update $\mathcal{M}$ to contain exactly the set of configurations which result from the empty sequence of processes linearizing, i.e., we leave $\mathcal{M}$ unchanged. Thus, a simple inspection of the augmentations to lines 2,5,6, and 7 reveals that our augmentation satisfies the intermediate line condition for a tracker.

3. Finally, in each return line, the definition of a tracker allows us to update $\mathcal{M}$ to contain any configurations C' that result from the return and an arbitrary sequence of processes with pending operations linearizing after any configuration $C \in \mathcal{M}$. Once again, we respect Observation 3, and choose all configurations C' that result from the return (and the empty sequence of processes linearizing thereafter) after each $C \in \mathcal{M}$. This is easily confirmed by inspecting the augmentations to lines 3 and 8.

By simple inspections, we have verified that the augmentation in Figure 11.7.2 is indeed a tracker of the union find implementation in Figure 11.7.1. Notably, we did not need to look at or understand the code of the original implementation (beyond mechanically verifying that it was copied-and-pasted to the augmentation properly). Even so, by Theorem 11.6.1, seeing a certification of the invariant $|\mathcal{M}| = 1$ from a mechanical proof verifier, suffices as a reliable and rigorous proof that the union-find implementation is Strongly Linearizable (and thereby also Linearizable)!

In the remainder of the section, we will describe *how* we went about proving that invariant.

11.7.4 Proving The Invariant

Our task is to prove prove that $\mathcal{I}_L \equiv \mathcal{M} \neq \varnothing$ and $\mathcal{I}_S \equiv |\mathcal{M}| = 1$ are invariants of $\mathcal{A}(\mathcal{O}')$ in order to deduce that $\mathcal{O}$ is linearizable and strongly linearizable. Obviously, $\mathcal{I}_S$ implies $\mathcal{I}_L$, so in this example we will focus on $\mathcal{I}_S$, i.e. strong linearizability, alone.

$\mathcal{I}_S$ clearly holds in the initial configuration of the algorithm, so it suffices to show that it will continue to hold in subsequent configurations; we will accomplish this task by induction. Of course, $\mathcal{I}_S$'s validity in subsequent configurations of the algorithm relies not only on its validity in the current state, but also on the design of the algorithm, i.e. other invariants of the algorithm that capture the states of the various program variables (objects and registers). Thus, in order to go through with our strategy, we must strengthen $\mathcal{I}_S$ to a stronger invariant $\mathcal{I}$ that meets two conditions: (1) $\mathcal{I}$ is *inductive* and (2) $\mathcal{I}$ implies $\mathcal{I}_S$. This task of strengthening $\mathcal{I}_S$ to an $\mathcal{I}$ that meets the conditions is the main intellectual work that the prover must do. Of course, the prover must subsequently *prove* that $\mathcal{I}$ is inductive and implies $\mathcal{I}_S$; but our experience through proving several algorithms suggests that this latter step, while potentially time-taking due to the length of the proof, is generally intellectually easier once the correct $\mathcal{I}$ is identified. In particular, the identification of $\mathcal{I}$ requires an understanding of "why the algorithm works", and thus the *prover* (*unlike the verifier*) must still, in general, understand the algorithm well in order to give the proof.

In the case of our union find implementation, we present the strengthened invariant $\mathcal{I}$ in Figure 11.7.3. $\mathcal{I}$ is a conjuction, and two of the conjuncts are $\mathcal{I}_S$ and $\mathcal{I}_L$. Thus, when we prove its an invariant, we clearly have $\mathcal{I}_S$ and $\mathcal{I}_L$ by implication. The remaining conjuncts can be understood as follows:

- The conjuncts $\mathcal{I}_{par}, \mathcal{I}_x, \mathcal{I}_y, \mathcal{I}_u, \mathcal{I}_v, \mathcal{I}_{pc}, \mathcal{I}_{\mathcal{M}}$ express type safety. That is, the various variables in the algorithm always take on values that we would expect satisfy their types. For instance, $\mathcal{I}_{par}$ expresses that for each node $z \in [n]$, $z.par$ is also a node.

- The conjucts $\mathcal{I}_a, \mathcal{I}_b, \mathcal{I}_c$ express general truths about our union find implementation. Namely, that every node z shares a part in the partition with its parent $z.par$; that if z and w are different roots, then they are different parts of the partition; and that parent pointers never point to smaller valued nodes.

- The conjuncts $\mathcal{I}_{1,4}, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_{5,6,7}, \mathcal{I}_8$ express truths that pertain to a process π when its program counter pc_π has a particular value. For instance, $\mathcal{I}_{5,6,7}$ expresses that if $pc_\pi \in \{5, 6, 7\}$, then

the pairs of elements x_π, u_π and y_π, v_π are each in the same part of the partition and that π is yet-to-linearize a UNITE operation with argument (x_π, y_π).

Figure 11.7.3: Invariant $\mathcal{I}$ of $\mathcal{A}(\overline{\mathcal{O}})$, where $\overline{\mathcal{O}}$ is the implementation of the union find tracker in Figure 11.7.2.

$$\begin{aligned}\mathcal{I} \equiv\ & \mathcal{I}_L \wedge \mathcal{I}_S \\ & \wedge\ \mathcal{I}_{par} \wedge \mathcal{I}_x \wedge \mathcal{I}_y \wedge \mathcal{I}_u \wedge \mathcal{I}_v \wedge \mathcal{I}_{pc} \wedge \mathcal{I}_{\mathcal{M}} \\ & \wedge\ \mathcal{I}_a \wedge \mathcal{I}_b \wedge \mathcal{I}_c \\ & \wedge\ \mathcal{I}_{1,4} \wedge \mathcal{I}_2 \wedge \mathcal{I}_3 \wedge \mathcal{I}_{5,6,7} \wedge \mathcal{I}_8\end{aligned}$$

In the above expression, the various conjuncts on the right hand side are defined below.

- $\mathcal{I}_L \equiv \mathcal{M} \neq \varnothing$
- $\mathcal{I}_S \equiv |\mathcal{M}| = 1$
- $\mathcal{I}_{par} \equiv \forall z \in [n] : z.par \in [n]$
- $\mathcal{I}_x \equiv \forall \pi \in \Pi : x_\pi \in [n]$
- $\mathcal{I}_y \equiv \forall \pi \in \Pi : y_\pi \in [n]$
- $\mathcal{I}_u \equiv \forall \pi \in \Pi : u_\pi \in [n]$
- $\mathcal{I}_v \equiv \forall \pi \in \Pi : v_\pi \in [n]$
- $\mathcal{I}_{pc} \equiv \forall \pi \in \Pi : pc_\pi \in [8]$
- $\mathcal{I}_{\mathcal{M}} \equiv \mathcal{M} \subseteq AtomicConfigurations$
- $\mathcal{I}_a \equiv \forall (\sigma, f) \in \mathcal{M} : \forall z \in [n] : z.par \in \sigma(z)$
- $\mathcal{I}_b \equiv \forall (\sigma, f) \in \mathcal{M} : \forall w, z \in [n] : (w \neq z \wedge w.par = w \wedge z.par = z) \implies \sigma(w) \neq \sigma(z)$
- $\mathcal{I}_c \equiv \forall z \in [n] : z.par \geq z$
- $\mathcal{I}_{1,4} \equiv \forall \pi \in \Pi : \forall (\sigma, f) \in \mathcal{M} : pc_\pi \in \{1, 4\} \implies f(\pi) = (\perp, \perp, \perp)$
- $\mathcal{I}_2 \equiv \forall \pi \in \Pi : \forall (\sigma, f) \in \mathcal{M} : pc_\pi = 2 \implies (\sigma(u_\pi) = \sigma(x_\pi) \wedge f(\pi) = (\textsc{Find}, x_\pi, \perp))$
- $\mathcal{I}_3 \equiv \forall \pi \in \Pi : \forall (\sigma, f) \in \mathcal{M} : pc_\pi = 3 \implies f(\pi) = (\textsc{Find}, x_\pi, u_\pi)$
- $\mathcal{I}_{5,6,7} \equiv \forall \pi \in \Pi : \forall (\sigma, f) \in \mathcal{M} : pc_\pi \in \{5, 6, 7\} \implies (\sigma(u_\pi) = \sigma(x_\pi) \wedge \sigma(v_\pi) = \sigma(y_\pi) \wedge f(\pi) = (\textsc{Unite}, (x_\pi, y_\pi), \perp))$
- $\mathcal{I}_8 \equiv \forall \pi \in \Pi : \forall (\sigma, f) \in \mathcal{M} : pc_\pi = 8 \implies f(\pi) = (\textsc{Unite}, (x_\pi, y_\pi), ack)$

Our intricate proof that $\mathcal{I}$ is indeed an inductive invariant of the system and proofs of the straightforward deductions that $\mathcal{I}$'s invariance implies the invariance of $\mathcal{I}_L$ and $\mathcal{I}_S$, thereby proving the strong linearizability of the union find implementation can be found at:

https://github.com/visveswara/machine-certified-linearizability/blob/master/UnionFindTracker.tla

11.8 Conclusion and Remarks

In this work, we have shown a novel technique by which machine-verified proofs of linearizability and strong linearizability can be produced for concurrent object implementations. In Theorems 11.4.4 and 11.6.1, we have proven the powerful guarantee that our technique is universal, sound, and complete. That is, any linearizable (resp. strongly linearizable) implementation—regardless of what type it implements and how complex the linearization structure is—can be proved linearizable (resp. strongly linearizable) via our techniques. Furthermore, our technique lends itself to machine-verified proofs. We have demonstrated this machine-verifiability by providing proofs of linearizability and strong linearizability of Jayanti and Tarjan's union-find implementation that have been certified correct by TLAPS. This result is particularly significant, since the union-find implementation has been noted for its speed and practical applicability [51, 95, 3]. In other works, we and our collaborators have used our technique to give machine-verified linearizability proofs of several other algorithms including the Herlihy-Wing queue, and Jayanti's single-writer single-scanner snapshot object [211, 210, 94]. We look forward to many more algorithms being successfully machine-verified, both by us and others in the community.

We are part-way through extending our proof technique to incorporate other variants of linearizability, such as *durable linearizability* for recoverable data structures designed for non-volatile random access memory (NVRAM). We believe an interesting related open problem is the development of such proof methods for consistency criteria that are weaker than linearizability, such as *sequential consistency*.

Part V

Machine Learning

Chapter 12

Hogwild Gibbs Sampling

12.1 Introduction

The increasingly ambitious applications of data analysis, and the corresponding growth in the size of the data that needs to processed has brought important scalability challenges to machine learning algorithms. Fundamental methods such as Gradient Descent and Gibbs sampling, which were designed with a sequential computational model in mind, are to be applied on datasets of increasingly larger size. As such, there has recently been increased interest towards developing techniques for parallelizing these methods. However, these algorithms are inherently sequential and are difficult to parallelize.

HOGWILD!-SGD, proposed by Niu et al. [165], is a lock-free asynchronous execution of stochastic gradient descent that has been shown to converge under the right sparsity conditions. Several variants of this method, and extensions of the asynchronous execution approach have been recently proposed, and have found successful applications in a broad range of applications ranging from PageRank approximation, to deep learning and recommender systems [213, 166, 158, 149, 143, 143, 49].

Similar to HOGWILD!-SGD, lock-free asynchronous execution of Gibbs sampling, called HOGWILD!-Gibbs, was proposed by Smola and Narayanamurthy [186], and empirically shown to work well on several models [214]. Johnson et al. [122] provide sufficient conditions under which they show theoretically that HOGWILD!-Gibbs produces samples with the correct mean in Gaussian models, while Terenin et al. [202] propose a modification to the algorithm that is shown to converge under some strong assumptions on asynchronous computation.

Algorithm 31 Gibbs Sampling

Input: Set of variables V, Configuration $x_0 \in S^{|V|}$, Distribution π

for $t = 1 \dots T$ **do**

Sample i uniformly from $\{1, 2, \dots, n\}$;

Sample $X_i \sim \Pr_\pi [.|X_{-i} = x_{-i}]$ and set $x_{i,t} = X_i$;

For all $j \neq i$, set $x_{j,t} = x_{j,t-1}$;

In a more recent paper, De Sa et al. [48] propose the study of HOGWILD!-Gibbs under a stochastic model of asynchronicity in graphical models with discrete variables. Whenever the graphical model satisfies Dobrushin's condition, they show that the mixing time of the asynchronous Gibbs sampler is similar to that of the sequential (synchronous) one. Moreover, they establish that the asynchronous Gibbs sampler accurately estimates probabilities of events on a sublinear number of variables, in particular events on up to $O(\varepsilon n/\log n)$ variables can be estimated within variational distance ε, where n is the total number of variables in the graphical model (Lemma 2, [48]).

Our Results. Our goal in this work is to push the theoretical understanding of HOGWILD!-Gibbs to estimate functions of *all the variables in a graphical model*. In particular, we are interested in whether HOGWILD!-Gibbs can be used to accurately estimate the expectations of such functions. Results from [48] imply that an accurate estimation is possible whenever the function under consideration is Lipschitz with a good Lipschitz constant with respect to the Hamming metric. Under the same Dobrushin condition used in [48] (see Definition 12.2.3), and under a stochastic model of asynchronicity with weaker assumptions (see Section 12.2.1), we show that you can do better than the bounds implied by [48] even for functions with bad Lipschitz constants. For instance, consider quadratic functions on an Ising model, which is a binary graphical model, and serves as a canonical example of Markov random fields [141, 159, 61, 47, 70, 58]. Under appropriate normalization, these functions take values in the range $[-n^2, n^2]$ and have a Lipschitz constant of n. Given this, the results of [48] would imply we can estimate quadratic functions on the Ising model within an error of $O(n)$. We improve this error to be of $O(\sqrt{n})$. In particular, we show the following in this chapter:

- Starting at the same initial configuration, the executions of the sequential and the asynchronous Gibbs samplers can be coupled so that the expected Hamming distance between the multivariate samples that the two samplers maintain is bounded by $O(\tau \log n)$, where n is the number of variables in the graphical model, and τ is a measure of the average contention

in the asynchronicity model of Section 12.2.1. See Lemma 12.3.1. More generally, the expectation of the d-th power of the Hamming distance is bounded by $C(d,\tau)\log^d n$, for some function $C(d,\tau)$. See Lemma 12.3.2.

- It follows from Lemmas 12.3.1 and 12.3.2 that, if a function f of the variables of a graphical model is K-Lipschitz with respect to the d-th power of the Hamming distance, then the bias in the expectation of f introduced by HOGWILD!-Gibbs under the asynchronicity model of Section 12.2.1 is bounded by $K \cdot C(d,\tau)\log^d n$. See Corollary 12.4.2.

- Next, we improve the bounds of Corollary 12.4.2 for functions that are degree-d polynomials of the variables of the graphical model. Low degree polynomials on graphical models are a natural class of functions which are of interest in many statistical tasks performed on graphical models (see, for instance, [46]). For simplicity we show these improvements for the Ising model, but our results are extendible to general graphical models. We show, in Theorem 12.4.4, that the bias introduced by HOGWILD!-Gibbs in the expectation of a degree-d polynomial of the Ising model is bounded by $O((n\log n)^{(d-1)/2})$. This bound improves upon the bound computed by Corollary 12.4.2 by a factor of about $(n/\log n)^{(d-1)/2}$, as the Lipschitz constant with respect to the Hamming distance of a degree-d polynomial of the Ising model can be up to $O(n^{d-1})$. Importantly, the bias of $O((n\log n)^{(d-1)/2})$ that we show is introduced by the asynchronicity is of a lower order of magnitude than the standard deviation of degree-d polynomials of the Ising model, which is $O((n)^{d/2})$—see Theorem 12.2.8, and which is already experienced by the sequential sampler. Moreover, in Theorem 12.4.7, we also show that the asynchronous Gibbs sampler is not adding a higher order variance to its sample. Thus, our results suggest that running Gibbs sampling asynchronously leads to a valid bias-variance tradeoff.

  Our bounds for the expected Hamming distance between the sequential and the asynchronous Gibbs samplers follow from coupling arguments, while our improvements for polynomial functions of Ising models follow from a combination of our Hamming bounds and recent concentration of measure results for polynomial functions of the Ising model [45, 71, 81].

- In Section 12.5, we illustrate our theoretical findings by performing experiments on a multicore machine. We experiment with graphical models over two kinds of graphs. The first is

the $\sqrt{n} \times \sqrt{n}$ grid graph (which we represent as a torus for degree regularity) where each node has 4 neighbors, and the second is the clique over n nodes.

We first study how valid the assumptions of the asynchronicity model are. The main assumption in the model was that the average contention parameter τ doesn't grow as the number of nodes in the graph grows. It is a constant which depends on the hardware being used and we observe that this is indeed the case in practice. The expected contention grows linearly with the number of processors on the machine but remains constant with respect to n (see Figures 12.5.1 and 12.5.2).

Next, we look at quadratic polynomials over graphical models associated with both the grid and clique graphs. We estimate their expected values under the sequential Gibbs sampler and HOGWILD!-Gibbs and measure the bias (absolute difference) between the two. Our theory predicts that this should scale at $\sqrt{n}$ and we observe that this is indeed the case (Figure 12.5.3). Our experiments are described in greater detail in Section 12.5.

12.2 The Model and Preliminaries

In this work, we consider the Gibbs sampling algorithm as applied to discrete graphical models. The models will be defined on a graph $G = (V, E)$ with $|V| = n$ nodes and will represent a probability distribution π. We use S to denote the range of values each node in V can take. For any configuration $X \in S^{|V|}$, $\pi_i(.|X^{-i})$ will denote the conditional distribution of variable i given all other variables of state X.

In Section 12.4, we will look at Ising models, a particular class of discrete binary graphical models with pairwise local correlations. We consider the Ising model on a graph $G = (V, E)$ with n nodes. This is a distribution over $\Omega = \{\pm 1\}^n$, with a parameter vector $\vec{\theta} \in \mathbb{R}^{|V|+|E|}$. $\vec{\theta}$ has a parameter corresponding to each edge $e \in E$ and each node $v \in V$. The probability mass function assigned to a string x is

$$P(x) = \exp\left(\sum_{v \in V} \theta_v x_v + \sum_{e=(u,v) \in E} \theta_e x_u x_v - \Phi(\vec{\theta})\right),$$

where $\Phi(\vec{\theta})$ is the log-partition function for the distribution. We say an Ising model has *no external field* if $\theta_v = 0$ for all $v \in V$. For ease of exposition we will focus on the case with no external field in this work. However, the results extend to Ising models with external fields when the functions

under consideration (in Section 12.4) are appropriately chosen to be *centered*. See [45].

Throughout this work we will focus on bounded functions defined on the discrete space $S^{|V|}$. For a function f, we use $\|f\|_\infty$ to denote the maximum absolute value of the function over its domain. We will use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. In Section 12.4, we will study polynomial functions over the Ising model. Since $x_i^2 = 1$ always in an Ising model, any polynomial function of degree d can be represented as a multilinear function of degree d and we will refer to them interchangeably in the context of Ising models.

Definition 12.2.1 (Polynomial/Multilinear Functions of the Ising Model)**.** A *degree-d polynomial* defined on n variables $x_1, \ldots, x_n$ is a function of the following form

$$\sum_{S \subseteq [n]: |S| \leq d} a_S \prod_{i \in S} x_i,$$

where $a : 2^{[n]} \to \mathbb{R}$ is a coefficient vector.

When the degree $d = 1$, we will refer to the function as a linear function, and when the degree $d = 2$ we will call it a bilinear function. Note that since $X_u \in \{\pm 1\}$, any polynomial function of an Ising model is a multilinear function. We will use a to denote the coefficient vector of such a multilinear function and $\|a\|_\infty$ to denote the maximum element of a in absolute value. Note that we will use permutations of the subscripts to refer to the same coefficient, i.e., a_{ijk} is the same as a_{jik}. Also we will use the term d-linear function to refer to a multilinear function of degree d.
At times, for simplicity we will instead consider degree d polynomials of the form

$$f_a(x) = \sum_{i_1 i_2, \ldots, i_d} a_{i_1 i_2 \ldots i_d} x_{i_1} x_{i_2} \ldots x_{i_d}$$

This form involves multiplicity in the terms, i.e. each monomial might appear multiple times. We can map from degree d polynomials without multiplicity of terms to functions of the above form in a straightforward manner by dividing each coefficient $a_{i_1 i_2 \ldots i_d}$ by an appropriate constant which captures how many times the term appears in the above notation. This constant lies between $d!$ and d^d.

We now give a formal definition of Dobrushin's uniqueness condition, also known as the high-temperature regime. First we define the influence of a node j on a node i.

Definition 12.2.2 (Influence in Graphical Models)**.** Let π be a probability distribution over some

set of variables V. Let B_j denote the set of state pairs (X, Y) which differ only in their value at variable j. Then the influence of node j on node i is defined as

$$I(j,i) = \max_{(X,Y)\in B_j} \left\|\pi_i(.|X^{-i}) - \pi_i(.|Y^{-i})\right\|_{TV}$$

Now, we are ready to state Dobrushin's condition.

Definition 12.2.3 (Dobrushin's Uniqueness Condition)**.** Consider a distribution π defined on a set of variables V. Let

$$\alpha = \max_{i\in V} \sum_{j\in V} I(j,i)$$

π is said to satisfy Dobrushin's uniqueness condition is $\alpha < 1$.

We have the following result from [48] about mixing time of Gibbs sampler for a model satisfying Dobrushin's condition.

Theorem 12.2.4 (Mixing Time of Sequential Gibbs Sampling)**.** *Assume that we run Gibbs sampling on a distribution that satisfies Dobrushin's condition, $\alpha < 1$. Then the mixing time of sequential-Gibbs is bounded by*

$$t_{mix-seq(\varepsilon)} \leq \frac{n}{1-\alpha} \log\left(\frac{n}{\varepsilon}\right).$$

Definition 12.2.5. For any discrete state space $S^{|V|}$ over the set of variables V, The *Hamming distance* between $x, y \in S^{|V|}$ is defined as $d_H(x,y) = \sum_{i\in V} \mathbb{1}_{\{x_i \neq y_i\}}$.

Definition 12.2.6 (The greedy coupling between two Gibbs Sampling chains)**.** Consider two instances of Gibbs sampling associated with the same discrete graphical model π over the state space $S^{|V|}$: $X_0, X_1, \ldots$ and $Y_0, Y_1, \ldots$. The following coupling procedure is known as the *greedy coupling*. Start chain 1 at X_0 and chain 2 at Y_0 and in each time step t, choose a node $v \in V$ uniformly at random to update in both the chains. Without loss of generality assume that $S = \{1, 2, \ldots, k\}$. Let $p(i_1)$ denote the probability that the first chain sets $X_{t,v} = i_1$ and let $q(i_2)$ be the probability that the second chain sets $Y_{t,v} = i_2$. Plot the points $\sum_{j=1}^{i} p(j) = P(i)$, and $\sum_{j=1}^{i} q(j) = Q(i)$ for all $i \in [k]$ on the interval from $[0, 1]$. Also pick $P(0) = Q(0) = 0$ and $P(k+1) = Q(k+1) = 1$. Couple the updates according to the following rule:

Draw a number x uniformly at random from $[0,1]$. Suppose $x \in [P(i_1), P(i_1+1)]$ and $x \in [Q(i_2), Q(i_2+1)]$. Choose $X_{t,v} = i_1$ and $Y_{t,v} = i_2$.

We state an important property of this coupling which holds under Dobrushin's condition, in the following Lemma.

Lemma 12.2.7. *The greedy coupling (Definition 12.2.6) satisfies the following property. Let $X_0, Y_0 \in S^{|V|}$ and consider two executions of Gibbs sampling associated with distribution π and starting at X_0 and Y_0 respectively. Suppose the executions were coupled using the greedy coupling. Suppose in the step $t = 1$, node i is chosen to be updated in both the models. Then,*

$$\Pr\left[X_{i,1} \neq Y_{i,1}\right] \leq \left\|\pi_i(.|X_0^{-i}) - \pi_i(.|Y_0^{-i})\right\|_{TV} \tag{12.2.1}$$

12.2.1 Modeling Asynchronicity

We use the asynchronicity model from [165] and [48]. Hogwild!-Gibbs is a multi-threaded algorithm where each thread performs a Gibbs update on the state of a graph which is stored in shared memory (typically in RAM). We view each processor's write as occuring at a distinct time instant. And each write starts the next time step for the process. Assuming that the writes are all serialized, one can now talk about the state of the system after t writes. This will be denoted as time t. HOGWILD! is modeled as a stochastic system adapted to a natural filtration $\mathcal{F}_t$. $\mathcal{F}_t$ contains all events thast have occured until time t. Some of these writes happen based on a read done a few steps ago and hence correspond to updates based on stale values in the local cache of the processor. The staleness is modeled in a stochastic manner using the random variable $\tau_{i,t}$ to denote the delay associated with the read performed on node i at time step t. The value of node i used in the update at time t is going to be $Y_{i,t} = X_{i,(t-\tau_{i,t})}$. Delays across different node reads can be correlated. However delay distribution is independent of the configuration of the model at time t. The model imposes two restrictions on the delay distributions. First, the expected value of each delay distribution is bounded by τ. We will think of τ as a constant compared to n in this work. We call τ the average contention parameter associated with a HOGWILD!-Gibbs execution. [48] impose a second restriction which bounds the tails of the distribution of $\tau_{i,t}$. We do not need to make this assumption in this work for our results. [48] need the assumption to show that the HOGWILD! chain mixes fast. However, by using coupling arguments we can avoid the need to have the HOGWILD! chain mix and will just use the mixing time bounds for the sequential Gibbs

sampling chain instead. Let T denote the set of all delay distributions. We refer to the sequential Gibbs sampler associated with a distribution π as G_π and the HOGWILD! Gibbs sampler together with T associated with a distribution p by H_p^{T}. Note that H_π is a time-inhomogenuous Markov chain and might not converge to a stationary distribution.

12.2.2 Properties of Polynomials on Ising Models satisfying Dobrushin's condition

Here we state some known results about polynomial functions on Ising models satisfying Dobrushin's condition.

Theorem 12.2.8 (Concentration of Measure for Polynomial Functions of the Ising model, [45, 71, 81])**.** *Consider an Ising model p without external field on a graph $G = (V, E)$ satisfying Dobrushin's condition with Dobrushin parameter $\alpha < 1$. Let f_a be a degree d-polynomial over the Ising model. Let $X \sim p$. Then, there is a constant $c(\alpha, \delta)$, such that,*

$$\Pr\left[|f_a(X) - \mathbf{E}\left[f_a(X)\right]| > t\right] \leq 2\exp\left(-\frac{(1-\alpha)t^{2/d}}{c(\alpha,d)\left\|a\right\|_\infty^{2/d} n}\right).$$

As a corollary this also implies,

$$\mathbf{Var}\left[f_a(X)\right] \leq C_3(d,\alpha)n^d.$$

Theorem 12.2.9 (Marginals Bound under Dobrushin's condition, [45])**.** *Consider an Ising model p satisfying Dobrushin's condition with Dobrushin parameter $\alpha < 1$. For some positive integer d, let $f_a(x)$ be a degree d polynomial function. Then we have that, if $X \sim p$,*

$$\left|\mathbf{E}\left[f_a(X)\right]\right| \leq 2\left(\frac{4nd\log n}{1-\alpha}\right)^{d/2}.$$

12.3 Bounding The Expected Hamming Distance Between Coupled Executions of HOGWILD! and Sequential Gibbs Samplers

In this Section, we show that under the greedy coupling of the sequential and asynchronous chains, the expected Hamming distance between the two chains at any time t is small. This will form the

basis for our accurate estimation results of Section 12.4. We begin by showing that the expected Hamming distance between the states X_t and Y_t of a coupled run of the sequential and asynchronous executions respectively, is bounded by a $(\tau\alpha \log n)/(1-\alpha)$. At a high level, the proof of Lemma 12.3.1 proceeds by studying the expected change in the Hamming distance under one step of the coupled execution of the chains. We can bound the expected change using the Dobrushin parameter and the property of the greedy coupling (Lemma 12.2.7). We then show that the expected change is negative whenever the Hamming distance between the two chains was above $O(\log n)$ to begin with. This allows us to argue that when the two chains start at the same configuration, then the expected Hamming distance remains bounded by $O(\log n)$.

Lemma 12.3.1. *Let π denote a discrete probability distribution on n variables (nodes) with Dobrushin parameter $\alpha < 1$. Let $G_\pi = X_0, X_1, \ldots, X_t, \ldots$ denote the execution of the sequential Gibbs sampler on π and $H_\pi^{\mathrm{T}} = Y_0, Y_1, \ldots, Y_t, \ldots$ denote the HOGWILD! Gibbs sampler associated with π such that $X_0 = Y_0$. Suppose the two chains are running coupled in a greedy manner. Let $\mathcal{K}_t$ denote all events that have occured until time t in this coupled execution. Then we have, for all $t \geq 0$, under the greedy coupling of the two chains,*

$$\mathbf{E}\left[d_H(X_t, Y_t)|\mathcal{K}_0\right] \leq \frac{\tau\alpha \log n}{1-\alpha}$$

Proof. We will show the statement of the Lemma is true by induction over t. The statement is clearly true at time $t = 0$ since $X_0 = Y_0$. Suppose it holds for some time $t > 0$. The state of the system at time t, $\mathcal{K}_t$ includes the choice of nodes the two chains chose to update at each time step, the delays $\tau_{i,t'}$ for each node i and time step $t' \leq t$ and the states $\{X_{t'}\}_{t \geq t' \geq 0}$ and $\{Y_{t'}\}_{t \geq t' \geq 0}$ under the two chains. We let $I_{t'}$ denote the node that was chosen to be updated in the two chains at time step t'. As a shorthand, we use L_t to denote $d_H(X_t, Y_t)$. We will first compute a bound on

$$\mathbf{E}\left[L_{t+1}|\mathcal{K}_t\right] \tag{12.3.1}$$

The induction hypothesis implies that $\mathbf{E}[L_t|\mathcal{K}_0] \leq \frac{\tau\alpha}{1-\alpha}$. Given the delays for time $t+1$, denote by Y_t' the following state

$$Y_{i,t}^{\tau} = Y_{i,t-\tau_{i,t+1}} \; \forall \, i.$$

We partition the set of nodes into the set where X_t and Y_t have the same value and the set where

they don't. Define $V_t^{=}$ and $V_t^{\neq}$ as follows.

$$V_t^{=} = \{i \in [n] \; s.t. \; X_{i,t} = Y_{i,t}\}$$
$$V_t^{\neq} = \{i \in [n] s.t. X_{i,t} \neq Y_{i,t}\}.$$

When the two samplers proceed to perform their update for time step $t+1$, in addition to the nodes in the set $V_t^{\neq}$ some additional nodes might appear to have different values under the asynchronous chain. This is because of the delays $\tau_{i,t+1}$. We use D_{t+1} to denote the set of nodes in $V_t^{=}$ whose values read by the asynchronous chain are different from those under the sequential chain.

$$D_{t+1} = \left\{ i \in [n] \;\middle|\; X_{i,t} \neq Y_{i,t-\tau_{i,t+1}} \right\} \tag{12.3.2}$$

Next, we proceed to obtain a bound on the expected size of D_{t+1} which we will use later. Let $\tau_{1,t}, \tau_{2,t}, \ldots, \tau_{n,t}$ denote the delays at time t. Let $\delta_{i,t}$ denote the last index before t when the node i's value was updated. Observe that the $\delta_{i,t}$s have to all be distinct. Then

$$\mathbf{E}\left[|D_{t+1}||\mathcal{K}_t\right] \leq \sum_{i \in V_t^{=}} \Pr\left[\tau_{i,t} > \delta_{i,t}\right] \leq \sum_{i \in V_t^{=}} \frac{\tau}{\delta_{i,t}} \leq \tau \left(\sum_{j=1}^{|V_t^{=}|} \frac{1}{i} \right) \leq \tau \log n. \tag{12.3.3}$$

Suppose a node i from the set $V_t^{=}$ was chosen at step $t+1$ to be updated in both the chains. Now, L_{t+1} is either L_t or $L_t + 1$. The probability that it is $L_t + 1$ is bounded above by the total variation between the corresponding conditional distributions.

$$\Pr\left[X_{i,t+1} \neq Y_{i,t+1} \mid \mathcal{K}_t,\; i \in V_t^{=} \text{ chosen in step } t+1\right] \leq \left\| \pi_i(.|X_t^{-i}) - \pi_i(.|Y_t^{\tau^{-i}}) \right\|_{TV} \tag{12.3.4}$$
$$\leq \sum_{j \in V_t^{\neq} \cup D_{t+1}} I(j,i). \tag{12.3.5}$$

where (12.3.4) follows from the property of the greedy coupling (Lemma 12.2.7) and (12.3.5) follows from the triangle inequality because total variation distance is a metric.

Now suppose that i was chosen instead from the set $V_t^{\neq}$. Now L_{t+1} is either L_t or $L_t - 1$. We

lower bound the probability that it is $L_t - 1$ using arguments similar to the above calculation.

$$\Pr\left[X_{i,t+1} = Y_{i,t+1} \middle| \mathcal{K}_t,\ i \in V_t^{\neq} \text{ was chosen in step } t+1\right] \geq 1 - \left\|\pi_i(.|X_t^{-i}) - \pi_i(.|Y_t^{\tau^{-i}})\right\|_{TV} \tag{12.3.6}$$

$$\geq 1 - \sum_{j \in V_t^{\neq} \cup D_{t+1}} I(j,i). \tag{12.3.7}$$

where (12.3.6) follows from the property of the greedy coupling (Lemma 12.2.7) and (12.3.7) follows from the triangle inequality because total variation distance is a metric.

Now the expected change in the Hamming distance

$$\mathbf{E}\left[L_{t+1} - L_t | \mathcal{K}_t\right] = \frac{1}{n}\mathbf{E}\left[\sum_{i \in V_t^{=}} \sum_{j \in V_t^{\neq} \cup D_{t+1}} I(j,i) - \sum_{i \in V_t^{\neq}} \left(1 - \sum_{j \in V_t^{\neq} \cup D_{t+1}} I(j,i)\right) | \mathcal{K}_t\right]$$

$$\leq \frac{1}{n}\mathbf{E}\left[\sum_{j \in V_t^{\neq} \cup D_{t+1}} \sum_{i \in V} I(j,i) - L_t \middle| \mathcal{K}_t\right] \leq \frac{\left(L_t + \mathbf{E}\left[|D_{t+1}||\mathcal{K}_t\right]\right)\alpha}{n} \leq \frac{(L_t + \tau \log n)\alpha - L_t}{n}.$$

where we used the fact that $\sum_{i \in V} I(j,i) \leq \alpha$ for any j.

Now,

$$\mathbf{E}\left[L_{t+1}|\mathcal{K}_0\right] = \mathbf{E}\left[\mathbf{E}\left[L_{t+1}|\mathcal{K}_t\right]|\mathcal{K}_0\right] \tag{12.3.8}$$

$$\leq \mathbf{E}\left[L_t + \frac{(L_t + \tau \log n)\alpha - L_t}{n} \middle| \mathcal{K}_0\right] \leq \frac{\tau\alpha \log n}{1-\alpha}\left(1 - \frac{1-\alpha}{n}\right) + \frac{\tau\alpha \log n}{n} \tag{12.3.9}$$

$$= \frac{\tau\alpha \log n}{1-\alpha}. \tag{12.3.10}$$

□

Next, we generalize the above Lemma to bound also the d^{th} moment of the Hamming distance between X_t and Y_t obtained from the coupled executions. The proof of Lemma 12.3.2 follows a similar flavor as that of Lemma 12.3.1. It is however more involved to bound the expected increase in the d^{th} power of the Hamming distance and it requires some careful analysis to see that the bound doesn't scale polynomially in n.

Lemma 12.3.2 (d^{th} moment bound on Hamming)**.** *Consider the same setting as that of Lemma 12.3.1. We have, for all $t \geq 0$, under the greedy coupling of the two chains,*

$$\mathbf{E}\left[d_H(X_t, Y_t)^d | \mathcal{K}_0\right] \leq C(\tau, \alpha, d) \log^d n,$$

where $C(.)$ is some function of the parameters τ, α and d.

Proof. Again we will employ the shorthand $L_t = d_H(X_t, Y_t)$. The proof is structured in the following way. We will show the statement of the Lemma is true by induction over t and d. To show the statement for a certain values of t, d we will use the inductive hypotheses of the statement for all t', d where $t' < t$ and for all t, d' where $d' < d$. Lemma 12.3.1 shows the statement for all values of t for $d = 1$. We will assume the statement holds for all t for $d' < d$.

Now, we proceed to show it is true for d. Clearly for $t = 0$ it holds because $d_H(X_0, Y_0)^d = 0$. Suppose it holds for some time $t > 0$. The state of the system at time t, $\mathcal{K}_t$ includes the choices of nodes the two chains chose to update at each time step, the delays $\tau_{i,t'}$ for each node i and time step $t' \leq t$ and the states $\{X_{t'}\}_{t \geq t' \geq 0}$ and $\{Y_{t'}\}_{t \geq t' \geq 0}$ under the two chains. We let $I_{t'}$ denote the node that was chosen to be updated in the two chains at time step t'. We will proceed in a similar way as we did in Lemma 12.3.1. We first compute a bound on

$$\mathbf{E}\left[L_{t+1}^d | \mathcal{K}_t\right] \tag{12.3.11}$$

as a function of L_t. The induction hypothesis implies that $\mathbf{E}[L_t^d | \mathcal{K}_0] \leq C(\tau, \alpha, d) \log^d n$. We partition the set of nodes into the set where X_t and Y_t have the same value and the set where they don't. Define $V_t^{=}$ and $V_t^{\neq}$ as follows.

$$V_t^{=} = \{i \in [n] \; s.t. \; X_{i,t} = Y_{i,t}\}$$
$$V_t^{\neq} = \{i \in [n] s.t. X_{i,t} \neq Y_{i,t}\}.$$

When the two samplers proceed to perform their update for time step $t + 1$, in addition to the nodes in the set $V_t^{\neq}$ some additional nodes might appear to have different values under the asynchronous chain. This is because of the delays $\tau_{i,t+1}$. We use D_{t+1} to denote the set of nodes in $V_t^{=}$ whose values read by the asynchronous chain are different from those under the sequential

chain.

$$D_{t+1} = \left\{ i \in [n] \;\middle|\; X_{i,t} \neq Y_{i,t-\tau_{i,t+1}} \right\} \tag{12.3.12}$$

Next, we use the bound on the expected size of D_{t+1} which was derived in Lemma 12.3.1.

$$\mathbf{E}\left[|D_{t+1}||\mathcal{K}_t\right] \leq \tau \log n. \tag{12.3.13}$$

Suppose a node i from the set $V_t^{=}$ was chosen at step $t+1$ to be updated in both the chains. Now, L_{t+1} is either L_t or $L_t + 1$. The probability that it is $L_t + 1$ is bounded above by the total variation between the corresponding conditional distributions.

$$\Pr\left[X_{i,t+1} \neq Y_{i,t+1} | \mathcal{K}_t,\ i \in V_t^{=} \text{ chosen in step } t+1\right] \leq \left\| \pi_i(.|X_t^{-i}) - \pi_i(.|Y_t^{\tau^{-i}}) \right\|_{TV} \tag{12.3.14}$$

$$\leq \sum_{j \in V_t^{\neq} \cup D_{t+1}} I(j,i). \tag{12.3.15}$$

(12.3.15) follows again due to the property of greedy coupling (Lemma 12.2.7) and the metric property of the total variation distance.

Now suppose that i was chosen instead from the set $V_t^{\neq}$. Now L_{t+1} is either L_t or $L_t - 1$. The probability that it is $L_t - 1$ is bounded below by the following.

$$\Pr\left[X_{i,t+1} = Y_{i,t+1} \middle| \mathcal{K}_t,\ i \in V_t^{\neq} \text{ chosen in step } t+1\right] \geq 1 - \left\| \pi_i(.|X_t^{-i}) - \pi_i(.|Y_t^{\tau^{-i}}) \right\|_{TV} \tag{12.3.16}$$

$$\geq 1 - \sum_{j \in V_t^{\neq} \cup D_{t+1}} I(j,i). \tag{12.3.17}$$

(12.3.15) follows again due to the property of greedy coupling (Lemma 12.2.7) and the metric property of the total variation distance.

Now the expected change in the value of the d^{th} power of the Hamming distance is

$$\mathbf{E}\left[L_{t+1}^d - L_t^d | \mathcal{K}_t\right] \tag{12.3.18}$$

$$= \frac{1}{n}\mathbf{E}\left[\sum_{i\in V_t^=}\sum_{j\in V_t^{\neq}\cup D_{t+1}} I(j,i)\left((L_t+1)^d - L_t^d\right) - \sum_{i\in V_t^{\neq}}\left(1 - \sum_{j\in V_t^{\neq}\cup D_{t+1}} I(j,i)\right)\left(L_t^d - (L_t-1)^d\right)|\mathcal{K}_t\right] \tag{12.3.19}$$

$$\leq \frac{1}{n}\left((L_t+1)^d - L_t^d\right)\mathbf{E}\left[\sum_{j\in V_t^{\neq}\cup D_{t+1}}\sum_{i\in V} I(j,i)\right] - \frac{L_t}{n}\left(L_t^d - (L_t-1)^d\right) \tag{12.3.20}$$

$$\leq \frac{1}{n}\left((L_t+1)^d - L_t^d\right)(L_t + \mathbf{E}\left[D_{t+1}|\mathcal{K}_t\right])\alpha - \frac{L_t}{n}\left(L_t^d - (L_t-1)^d\right) \tag{12.3.21}$$

$$\leq \frac{1}{n}\left((L_t+1)^d - L_t^d\right)(L_t + \tau\log n)\alpha - \frac{L_t}{n}\left(L_t^d - (L_t-1)^d\right) \tag{12.3.22}$$

Now, suppose $\mathbf{E}\left[L_t^d \middle| \mathcal{K}_0\right] \leq C(\tau,\alpha,d)\log^d n - c_3(d)C(\tau,\alpha,d-1)\log^{d-1} n$ for an appropriate constant $c_3(d)$.Then,

$$\mathbf{E}\left[L_{t+1}^d - L_t^d \middle| \mathcal{K}_0\right] = \mathbf{E}\left[(L_{t+1} - L_t)\left(\sum_{i=0}^{d-1} L_{t+1}^d - i - 1 L_t^i\right)\right] \tag{12.3.23}$$

$$\leq \mathbf{E}\left[\sum_{i=0}^{d-1} L_{t+1}^{d-i-1} L_t^i\right] \leq c_3(d)C(\tau,\alpha,d-1)\log^{d-1} n \tag{12.3.24}$$

$$\implies \mathbf{E}\left[L_{t+1}^d\right] \leq C(\tau,\alpha,d)\log^d n, \tag{12.3.25}$$

where (12.3.24) follows because

$$\mathbf{E}\left[L_{t+1}^{d-i-1}L_t^i\right] \leq \mathbf{E}\left[L_t^i(L_t+1)^{d-i-1}\right] \leq \mathbf{E}\left[L_t^{d-1}\right] + c(d)o\left(\mathbf{E}\left[L_t^{d-1}\right]\right). \tag{12.3.26}$$

Suppose instead that $\mathbf{E}\left[L_t^d|\mathcal{K}_0\right]$ was larger than $C(\tau,\alpha,d)\log^d n - c_3(d)C(\tau,\alpha,d-1)\log^{d-1} n$. We want to show that for $C(\tau,\alpha,d)$ appropriately large in d, $\mathbf{E}\left[L_{t+1}^d|\mathcal{K}_0\right]$ doesn't exceed $C(\tau,\alpha,d)$.

$$\mathbf{E}\left[L_{t+1}^d|\mathcal{K}_0\right] = \mathbf{E}\left[\mathbf{E}\left[L_{t+1}^d\middle|\mathcal{K}_t\right]\middle|\mathcal{K}_0\right] \tag{12.3.27}$$

$$\leq \mathbf{E}\left[L_t^d + \frac{(L_t+\tau\log n)\alpha}{n}\left((L_t+1)^d - L_t^d\right) - \frac{L_t}{n}\left(L_t^d - (L_t-1)^d\right)\middle|\mathcal{K}_0\right] \tag{12.3.28}$$

$$= \mathbf{E}\left[L_t^d + \frac{(L_t+\tau\log n)\alpha}{n}\left(\sum_{i=1}^{d}\binom{d}{i}L_t^{d-i}\right) - \frac{L_t}{n}\left(\sum_{i=1}^{d}\binom{d}{i}L_t^{d-i}(-1)^i\right)\middle|\mathcal{K}_0\right] \tag{12.3.29}$$

$$\leq C(\tau,\alpha,d)\log^d n - \frac{1}{n}\left(\mathbf{E}[L_t^d](1-\alpha) - \sum_{i=2}^{d}\binom{d}{i}L_t^{d-i+1}(1+\alpha) - \sum_{i=1}^{d}\binom{d}{i}L_t^{d-i}\tau\alpha\log n\right) \tag{12.3.30}$$

$$\leq C(\tau,\alpha,d)\log^d n, \tag{12.3.31}$$

where (12.3.31) holds because for $C(\tau,\alpha,d)$ sufficiently large compared to the values of $C(\tau,\alpha,d')$, where $d' < d$, we can have $C(\tau,\alpha,d)\log^d n - c_3(d)C(\tau,\alpha,d-1)\log^{d-1} n$ dominating all the remaining terms in the two summations. Hence, by induction we have the desired Lemma statement. □

12.4 Estimating Global Functions Using HOGWILD! Gibbs Sampling

To begin with, we observe that our Hamming moment bounds from Section 12.3 imply that we can accurately estimate functions or events of the graphical model if they are Lipschitz. We show this below as a Corollary of Lemma 12.3.2. Before we state the Corollary, we will first state the following simple Lemma which quantifies how large a t is required to have an accurate estimate from the Gibbs sampler.

Lemma 12.4.1. *Let π be an graphical model on n nodes satisfying Dorbushin's condition with Dobrushin parameter $\alpha < 1$. Let $X_0, X_1, \ldots, X_t$ denote the steps of a Gibbs sampler running on π. Let $X \sim \pi$. Also let $f(x)$ be a bounded function on the graphical model. Then for $t > 0$,*

$$|\mathbf{E}[f_a(X_t)] - \mathbf{E}[f_a(X)]| \leq \|f\|_\infty n \exp\left(-\frac{(1-\alpha)t}{n}\right).$$

Proof. We have from Theorem 12.2.4 that $d_{\mathrm{TV}}(X_t, X) \leq n \exp\left(\frac{(1-\alpha)t}{n}\right)$. This implies

$$|\mathbf{E}[f_a(X_t)] - \mathbf{E}[f_a(X)]| \leq \left|\max_x f_a(x)\right| n \exp\left(\frac{(1-\alpha)t}{n}\right) \leq \|a\|_\infty n^{d+1} \exp\left(-\frac{(1-\alpha)t}{n}\right). \quad (12.4.1)$$

□

Now, we state Corollary 12.4.2 which quantifies the error we can attain when trying to estimate expectations of Lipschitz functions using HOGWILD!-Gibbs.

Corollary 12.4.2. *Let π denote the distribution associated with a graphical model over the set of variables V ($|V| = n$) taking values in a discrete space S^n. Assume that the model satisfies Dobrushin's condition with Dobrushin parameter $\alpha < 1$. Let $f : S^{|V|} \to \mathbb{R}$ be a function such that, for all $x, y \in S^{|V|}$,*

$$|f(x) - f(y)| \leq K d_H(x, y)^d.$$

Let $X \sim \pi$ and let $Y_0, Y_1, \ldots, Y_t$ denote an execution of HOGWILD!-Gibbs sampling on π with average contention parameter τ. For $t > \frac{n}{1-\alpha} \log(2 \|f\|_\infty n/K)$,

$$|\mathbf{E}[f(Y_t)] - \mathbf{E}[f(X)]| \leq K.(C(\tau, \alpha, d) \log^d n + 1).$$

where $C(.)$ is the function from Lemma 12.3.2.

Proof. Consider an execution $X_0, X_1, \ldots, X_t$, where $X_0 = Y_0$, of the synchronous Gibbs sampling associated with π coupled greedily with the HOGWILD! chain. We have from 12.2.4 and 12.4.1 that, for $t > \frac{n}{1-\alpha} \log(2 \|f\|_\infty n/K)$,

$$|\mathbf{E}[X_t] - \mathbf{E}[X]| \leq K. \quad (12.4.2)$$

Next, we have,

$$\mathbf{E}[f(X_t) - f(Y_t)] \leq \mathbf{E}\left[K d_H(X_t, Y_t)^d\right] \leq K.C(\tau, \alpha, d) \log^d n. \quad (12.4.3)$$

Putting (12.4.2) and (12.4.3) together we get the statement of the Corollary. □

We note that the results of [48] can be used to obtain Corollary 12.4.2 when the function is

Lipschitz with respect to the Hamming distance. The above corollary provides a simple way to bound the bias introduced by HOGWILD! in estimation of Lipschitz functions. However, many functions of interest over graphical models are not Lipschitz with good Lipschitz constants. In many cases, even when the Lipschitz constants are bad, there is still hope for more accurate estimation. As it turns out Dobrushin's condition provides such cases. We will focus on one such case which is polynomial functions of the Ising model. Our goal will be to accurately estimate the expected values of constant degree polynomials over the Ising model. Using the bounds from Lemmas 12.3.1 and 12.3.2, we now proceed to bound the bias in computing polynomial functions of the Ising model using HOGWILD! Gibbs sampling.

We first remark that linear functions (degree 1 polynomials) suffer 0 bias in their expected values due to HOGWILD!-Gibbs. This is because under zero external field Ising models $\mathbf{E}[\sum_i a_i X_i] = 0$ since each node individually has equal probability of being ± 1. This symmetry is maintained by HOGWILD!-Gibbs since the delays are configuration-agnostic. Hence the delays when a node is $+1$ and when it is -1 can be coupled perfectly leaving the symmetry intact. More interesting cases start happening when we go to degree 2 polynomials. Therefore, we start our investigation at quadratic polynomials. Theorem 12.4.3 states the bound we show for the bias in computation of degree 2 polynomials of the Ising model.

Theorem 12.4.3 (Bias in Quadratic functions of Ising Model computed using HOGWILD!-Gibbs)**.** *Consider the quadratic function $f_a(x) = \sum_{i,j:i<j} a_{ij} x_i x_j$. Let p denote an Ising model on n nodes with Dobrushin parameter $\alpha < 1$. Let $\{X_t\}_{t\geq 0}$ denote a run of sequential Gibbs sampler and $H_p^{\mathrm{T}} = \{Y_t\}_{t\geq 0}$ denote a run of HOGWILD!- Gibbs on p, such that $X_0 = Y_0$. Then we have, for $t > \frac{6n}{1-\alpha} \log(2 \|a\|_\infty n)$, under the greedy coupling of the two chains,*

$$|\mathbf{E}[f_a(X_t) - f_a(Y_t)]| \leq c_2 \|a\|_\infty \frac{\tau\alpha \log n}{(1-\alpha)^{3/2}} (n \log n)^{1/2}.$$

Proof. Under the greedy coupling, we have that,

$$|\mathbf{E}[f_a(X_t) - f_a(Y_t)]| = \tag{12.4.4}$$

$$\left| \mathbf{E}\left[\sum_{i, X_{i,t} \neq Y_{i,t}} \sum_{j > i, X_{j,t} = Y_{j,t}} a_{ij} \left(X_{i,t} X_{j,t} - Y_{i,t} Y_{j,t} \right) \right] + \mathbf{E}\left[\sum_{i, X_{i,t} = Y_{i,t}} \sum_{j > i, X_{j,t} \neq Y_{j,t}} a_{ij} \left(X_{i,t} X_{j,t} - Y_{i,t} Y_{j,t} \right) \right] \right| \tag{12.4.5}$$

$$= \left| \mathbf{E}\left[\sum_{i} \left(X_{i,t} - Y_{i,t} \right) \sum_{j > i, X_{j,t} = Y_{j,t}} a_{ij} X_{j,t} \right] + \mathbf{E}\left[\sum_{i, X_{i,t} = Y_{i,t}} X_{i,t} \sum_{j > i, X_{j,t} \neq Y_{j,t}} a_{ij} \left(X_{j,t} - Y_{j,t} \right) \right] \right| \tag{12.4.6}$$

$$\begin{aligned} &= \left| \mathbf{E}\left[\sum_{i} \left(X_{i,t} - Y_{i,t} \right) \sum_{j > i} a_{ij} X_{j,t} \right] - \mathbf{E}\left[\sum_{i} \left(X_{i,t} - Y_{i,t} \right) \sum_{j > i, X_{j,t} \neq Y_{j,t}} a_{ij} X_{j,t} \right] \right. \\ &\left. + \mathbf{E}\left[\sum_{i} X_{i,t} \sum_{j > i} a_{ij} \left(X_{j,t} - Y_{j,t} \right) \right] - \mathbf{E}\left[\sum_{i, X_{i,t} \neq Y_{i,t}} X_{i,t} \sum_{j > i} a_{ij} \left(X_{j,t} - Y_{j,t} \right) \right] \right| \end{aligned} \tag{12.4.7}$$

$$= \left| \mathbf{E}\left[\sum_{i} (X_{i,t} - Y_{i,t}) \sum_{j} a_{ij} X_{j,t} \right] + \mathbf{E}\left[\sum_{i} (X_{i,t} - Y_{i,t}) \sum_{j, X_{j,t} \neq Y_{j,t}} X_{j,t} \right] \right| \tag{12.4.8}$$

$$\leq \left| \mathbf{E}\left[\sum_{i} (X_{i,t} - Y_{i,t}) \sum_{j} a_{ij} X_{j,t} \right] \right| + \mathbf{E}\left[d_H(X_t, Y_t)^2 \right]. \tag{12.4.9}$$

where (12.4.5) is based on the observation that if $X_{i,t} X_{j,t} = Y_{i,t} Y_{j,t}$ then the difference associated with this monomial vanishes, (12.4.7) and (12.4.8) follow via rearrangement of terms, and (12.4.9) follows because $\left| \sum_{j, X_{j,t} \neq Y_{j,t}} X_{j,t} \right| \leq d_H(X_t, Y_t)$.

We bound each term in (12.4.9) seperately. First let us consider the term $\left| \mathbf{E}\left[\sum_i (X_{i,t} - Y_{i,t}) \sum_j a_{ij} X_{j,t} \right] \right|$. We will employ concentration of measure of linear functions of the Ising model to bound this term. Intuitively when t is large enough, X_t is very close to a sample from the true Ising model and hence X_t will have the properties of a true sample from the Ising model. In particular, we can employ

Theorem 12.2.8 to argue that if $X \sim p$, then

$$\Pr\left[\left|\sum_j a_{ij} X_j\right| > r\right] \leq 2\exp\left(-\frac{r^2(1-\alpha)}{8\left\|a\right\|_\infty^2 n}\right) \ \forall\, i$$

$$\implies \Pr\left[\left|\sum_j a_{ij} X_j\right| \leq c\left\|a\right\|_\infty \sqrt{\frac{n\log n}{1-\alpha}}\ \forall\, i\right] \geq 1 - \frac{1}{n^3} \tag{12.4.10}$$

$$\implies \Pr\left[\left|\sum_j a_{ij} X_{j,t}\right| \leq c\left\|a\right\|_\infty \sqrt{\frac{n\log n}{1-\alpha}}\ \forall\, i\right] \geq 1 - \frac{2}{n^3} \tag{12.4.11}$$

where (12.4.10) holds for a large enough constant c and (12.4.11) holds via an application of Lemma 12.4.1 for bilinear functions of the Ising model on the fact that $t > \frac{6n}{1-\alpha}\log(2\left\|a\right\|_\infty n)$. Denote by the set G_i, the following set of states

$$G_i = \left\{x \in \Omega \middle| \left|\sum_j a_{ij} x_j\right| \leq c\left\|a\right\|_\infty \sqrt{\frac{n\log n}{1-\alpha}}\right\} \tag{12.4.12}$$

Now,

$$\mathbf{E}\left[\sum_i (X_{i,t} - Y_{i,t}) \sum_j a_{ij} X_{j,t}\right] \tag{12.4.13}$$

$$\leq \mathbf{E}\left[\sum_i (X_{i,t} - Y_{i,t})(c\left\|a\right\|_\infty \sqrt{\frac{n\log n}{1-\alpha}}) \middle| \forall\, i, X_t \in G_i\right] + \left\|a\right\|_\infty n^2 \Pr\left[\exists\, i :\ X_t \notin G_i\right] \tag{12.4.14}$$

$$\leq c\left\|a\right\|_\infty \sqrt{\frac{n\log n}{1-\alpha}} \mathbf{E}\left[d_H(X_t, Y_t) | \forall\, i\ X_t \in G_i\right] + \frac{2}{n^3}. \tag{12.4.15}$$

where in (12.4.14) we used the fact that $\sum_i (X_i - Y_i)\sum_j a_{ij} X_j \leq \left\|a\right\|_\infty n^2$ and (12.4.15) follows because . Now,

$$\mathbf{E}\left[d_H(X_t, Y_t) | \forall\, i\ X_t \in G_i\right] \leq \frac{\mathbf{E}\left[d_H(X_t, Y_t)\right]}{\Pr\left[\forall\, i\ X_t \in G_i\right]} \leq 2\mathbf{E}\left[d_H(X_t, Y_t)\right] \leq \frac{2\tau\alpha\log n}{1-\alpha}, \tag{12.4.16}$$

where we have used that $\Pr[\forall\, i\ X_t \in G_i] \geq 1 - 2/n^3 \geq 1/2$ and employed Lemma 12.3.1. Hence, $\left|\mathbf{E}\left[\sum_i (X_{i,t} - Y_{i,t})\sum_j a_{ij} X_{j,t}\right]\right| \leq (c+1)\left\|a\right\|_\infty \sqrt{\frac{n\log n}{1-\alpha}}\frac{2\tau\alpha\log n}{1-\alpha}$. The second term we need to bound is

$$\mathbf{E}\left[d_H(X_t, Y_t)^2\right] \leq C(\tau, \alpha, 2)\log^2 n \tag{12.4.17}$$

which follows from Lemma 12.3.2. Putting the two bounds together we get the statement of the Theorem. □

The main intuition behind the proof is that we can improve upon the bound implied by the Lipschitz constant by appealing to strong concentration of measure results about functions of graphical models under Dobrushin's condition [45, 71, 81].

We extend the ideas in the above proof to bound the bias introduced by the HOGWILD!-Gibbs algorithm when computing the expected values of a degree d polynomial of the Ising model in high temperature. Our main result concerning d-linear functions is Theorem 12.4.4.

Theorem 12.4.4 (Bias in degree d polynomials computed using HOGWILD!-Gibbs)**.** *Consider a degree d polynomial of the form $f_a(x) = \sum_{i_1 i_2, \ldots, i_d} a_{i_1 i_2 \ldots i_d} x_{i_1} x_{i_2} \ldots x_{i_d}$. Consider the same setting as that of Theorem 12.4.3. Then we have, for $t > \frac{n(d+1)}{1-\alpha} \log n$, under the greedy coupling of the two chains,*

$$|\mathbf{E}[f_a(X_t) - f_a(Y_t)]| \leq c' \|a\|_\infty (n \log n)^{(d-1)/2}.$$

To show Theorem 12.4.4, we will use the following helper Lemmas: 12.4.5 and 12.4.6. We will state them first.

For simplicity, here we consider degree d polynomials of the form $f_a(x) = \sum_{i_1 i_2, \ldots, i_d} a_{i_1 i_2 \ldots i_d} x_{i_1} x_{i_2} \ldots x_{i_d}$.

Lemma 12.4.5. *Consider a degree d polynomial $f_a(x) = \sum_{i_1 i_2, \ldots, i_d} a_{i_1 i_2 \ldots i_d} x_{i_1} x_{i_2} \ldots x_{i_d}$. Let p denote a high temperature Ising model on a graph $G = (V, E)$ with $|V| = n$ nodes with Dobrushin parameter $\alpha < 1$. Let $G_p = X_0, X_1, \ldots, X_t, \ldots$ denote the synchronous Gibbs sampler on p and $H_p^{\mathrm{T}} = Y_0, Y_1, \ldots, Y_t, \ldots$ denote the HOGWILD! Gibbs sampler associated with p such that $X_0 = Y_0$. Then we have, for $t > \frac{n(d+1)}{1-\alpha} \log n$, under the greedy coupling of the two chains, for all $0 \leq k \leq d$,*

$$\left| \mathbf{E} \left[\sum_{i_1} (X_{i_1,t} - Y_{i_1,t}) \left(\ldots \left(\sum_{i_k} (X_{i_k,t} - Y_{i_k,t}) \sum_{i_{k+1}, \ldots, i_d} a_{i_1 \ldots i_d} \prod_{l=k+1}^{d} X_{i_l} \right) \right) \right] \right|$$
$$\leq C(\tau, \alpha, k) \log^k n C_2(\alpha, d-k) (n \log n)^{(d-k)/2}.$$

Proof. We will employ the bound we have on the moments on the Hamming distance 12.3.2 together with concentration of measure for polynomial functions of the Ising model 12.2.8 to show the statement. We have from Theorems 12.2.8 and 12.2.9 and Lemma 12.4.1, that for every choice of

$i_1, i_2, \dots i_k \in V$, and for $t > \frac{n(d+1)}{1-\alpha} \log n$,

$$\Pr\left[\left|\sum_{i_{k+1},\dots,i_d} a_{i_1\dots i_d} \prod_{l=k+1}^{d} X_{i_l,t}\right| > c_4(\alpha, d-k)\left(\frac{\|a\|_\infty n \log n}{1-\alpha}\right)^{(d-k)/2}\right] \leq \frac{1}{n^{d+2}\|a\|_\infty} \tag{12.4.18}$$

$$\implies \Pr\left[\exists\{i_1, i_2, \dots i_k \in V\} \left|\sum_{i_{k+1},\dots,i_d} a_{i_1\dots i_d} \prod_{l=k+1}^{d} X_{i_l,t}\right| > c_4(\alpha, d-k)\left(\frac{n \log n}{1-\alpha}\right)^{(d-k)/2}\right] \tag{12.4.19}$$

$$\leq \frac{1}{n^{d-k+2}\|a\|_\infty}, \tag{12.4.20}$$

where we used the fact that when t is large, the distribution of X_t is close to p (Lemma 12.4.1) and (12.4.19) follows from a union bound. Define the set of states $G_{i_1 i_2 \dots i_k}$ as follows.

$$G_{i_1 i_2 \dots i_k} = \left\{ x \in \Omega \left| \left| \sum_{i_{k+1},\dots,i_d} a_{i_1\dots i_d} \prod_{l=k+1}^{d} x_{i_l} \right| \leq c_4(\alpha, d-k)\left(\frac{n \log n}{1-\alpha}\right)^{(d-k)/2} \forall\{i_1, i_2, \dots i_k \in V\} \right.\right\} \tag{12.4.21}$$

Then we have,

$$\left|\mathbf{E}\left[\sum_{i_1}(X_{i_1,t} - Y_{i_1,t})\left(\dots\left(\sum_{i_k}(X_{i_k,t} - Y_{i_k,t})\sum_{i_{k+1},\dots,i_d} a_{i_1\dots i_d}\prod_{l=k+1}^{d} X_{i_l,t}\right)\right)\right]\right| \tag{12.4.22}$$

$$\leq \mathbf{E}\left[\sum_{i_1}|X_{i_1,t} - Y_{i_1,t}|\left(\dots\left(\sum_{i_k}|X_{i_k,t} - Y_{i_k,t}|\left|\sum_{i_{k+1},\dots,i_d} a_{i_1\dots i_d}\prod_{l=k+1}^{d} X_{i_l,t}\right|\right)\right)\right] \tag{12.4.23}$$

$$\leq c_4(\alpha, d-k)\left(\frac{n \log n}{1-\alpha}\right)^{(d-k)/2} \mathbf{E}\left[\left(\sum_i |X_{i,t} - Y_{i,t}|\right)^k \middle| X_t \in G_{i_1\dots i_k} \forall\{i_1, \dots, i_k \in V\}\right] + \frac{1}{n^2} \tag{12.4.24}$$

$$\leq c_4(\alpha, d-k)\left(\frac{n \log n}{1-\alpha}\right)^{(d-k)/2} 2^k \mathbf{E}\left[d_H(X_t, Y_t)^k \,\middle|\, X_t \in G_{i_1\dots i_k} \forall\{i_1, \dots, i_k \in V\}\right] + \frac{1}{n^2} \tag{12.4.25}$$

$$\leq c_4(\alpha, d-k)\left(\frac{n \log n}{1-\alpha}\right)^{(d-k)/2} 2^k.2C(\tau, \alpha, k)\log^k n \leq C(\tau, \alpha, k)\log^k n C_2(\alpha, d-k)(n \log n)^{(d-k)/2}. \tag{12.4.26}$$

where we have used the fact that $\sum_i |X_i - Y_i| = 2d_H(X, Y)$ for $X, Y \in \Omega$, and (12.4.26) follows from Lemma 12.3.2 and the observation that $\Pr\left[X_t \in G_{i_1\dots i_k} \forall\{i_1, \dots, i_k \in V\}\right] \geq 1 - \frac{1}{n^{d-k+2}\|a\|_\infty} \geq$

1/2. □

Lemma 12.4.6. *Consider a degree d polynomial $f_a(x) = \sum_{i_1 i_2, \ldots, i_d} a_{i_1 i_2 \ldots i_d} x_{i_1} x_{i_2} \ldots x_{i_d}$. Let p denote a high temperature Ising model on a graph $G = (V, E)$ with $|V| = n$ nodes with Dobrushin parameter $\alpha < 1$. Let $G_p = X_0, X_1, \ldots, X_t, \ldots$ denote the synchronous Gibbs sampler on p and $H_p^{\mathrm{T}} = Y_0, Y_1, \ldots, Y_t, \ldots$ denote the HOGWILD! Gibbs sampler associated with p such that $X_0 = Y_0$. Then we have, for $t > \frac{n(d+1)}{1-\alpha} \log n$, under the greedy coupling of the two chains, for all $0 \leq k \leq d$,*

$$\left| \mathbf{E} \left[\sum_{i_1} (X_{i_1,t} - Y_{i_1,t}) \left(\ldots \left(\sum_{i_k} (X_{i_k,t} - Y_{i_k,t}) \sum_{i_{k+1}, \ldots, i_d} a_{i_1 \ldots i_d} \left(\prod_{l=k+1}^{d} X_{i_l,t} - \prod_{l=k+1}^{d} Y_{i_l,t} \right) \right) \right) \right] \right|$$
$$\leq C(\tau, \alpha, k) \log^k n C_2(\alpha, d-k) (n \log n)^{(d-k)/2}.$$

Proof. We prove this by inducting backwards on k. When $k = d$, we get the desired statement from Lemma 12.3.2. Assume the statement holds for some $k + 1 < d$. We will show it holds for k as well. In the following calculations, we will, for a while, drop the subscript t from X_t and Y_t for brevity. We have,

$$\left| \mathbf{E} \left[\sum_{i_1} (X_{i_1} - Y_{i_1}) \left(\ldots \left(\sum_{i_k} (X_{i_k} - Y_{i_k}) \sum_{i_{k+1}, \ldots, i_d} a_{i_1 \ldots i_d} \left(\prod_{l=k+1}^{d} X_{i_l} - \prod_{l=k+1}^{d} Y_{i_l} \right) \right) \right) \right] \right| \tag{12.4.27}$$
$$= \left| \mathbf{E} \left[\sum_{i_1} (X_{i_1} - Y_{i_1}) \left(\ldots \left(\sum_{i_k} (X_{i_k} - Y_{i_k}) \sum_{\substack{i_{k+1} \ldots i_d \\ X_{i_{k+1}} \ldots X_{i_d} \neq Y_{i_{k+1}} \ldots Y_{i_d}}} a_{i_1 \ldots i_d} \left(2 \prod_{l=k+1}^{d} X_{i_l} \right) \right) \right) \right] \right|. \tag{12.4.28}$$

The last summation of (12.4.28) is over indices $i_{k+1}, \ldots, i_d$ such that the product of values in X at these indices disagrees with the corresponding product of values in Y. This can happen due to a disagreement between X and Y at one of the indices $i_{k+1}, \ldots, i_d$, say j, and an agreement of the product over the rest of the indices. That is, $X_j \neq Y_j$ and $\prod_{l=k+1}^{j-1} X_{i_j} \prod_{l=j+1}^{d} X_{i_j} = \prod_{l=k+1}^{j-1} Y_{i_j} \prod_{l=j+1}^{d} Y_{i_j}$. This leads to the last summation in (12.4.28) being bounded above by the sum of $d - k$ terms of the form

$$\sum_{i_j} (X_{i_j} - Y_{i_j}) \sum_{i_{k+1}, \ldots, i_{j-1}, i_{j+1}, \ldots, i_d} a_{i_1 \ldots i_d} \left(\prod_{l=k+1}^{j-1} X_{i_l} \prod_{l=j+1}^{d} X_{i_l} - \prod_{l=k+1}^{j-1} Y_{i_l} \prod_{l=j+1}^{d} Y_{i_l} \right). \tag{12.4.29}$$

Replacing the last summation of (12.4.28) with $d-k$ terms of the above form we see that the whole quantity decomposes into $d-k$ terms such that the inductive hypothesis can be applied over each of the terms. It is a fairly simple calculation to see that then we get the desired bound of the Theorem by induction (by keeping in mind that the quantity of focus here is the dependence on n and we treat d as a constant). □

Given Lemma 12.4.6, Theorem 12.4.4 follows. *Proof of Theorem 12.4.4:* The Theorem follows directly from Lemma 12.4.6 applied to the case $k = 0$. □

Next, we show that we can accurately estimate the expectations above by showing that the variance of the functions under the asynchronous model is comparable to that of the functions under the sequential model.

Theorem 12.4.7 (Variance of degree d polynomials computed using HOGWILD!-Gibbs)**.** *Consider a high temperature Ising model p on n nodes with Dobrushin parameter $\alpha < 1$. Let $f_a(x)$ be a degree d polynomial function Let $Y_0, Y_1, \ldots, Y_t$ denote a run of HOGWILD! Gibbs sampling associated with p. We have, for $t > \frac{(d+1)n}{1-\alpha} \log\left(n^2\right)$,*

$$\mathbf{Var}\left[f(Y_t)\right] \leq \|a\|_\infty^2 \, C(d, \alpha, \tau) n^d.$$

Proof.

$$\mathbf{Var}\left[f_a(Y_t)\right] = \mathbf{E}\left[f_a(Y_t)^2\right] - \mathbf{E}\left[f_a(Y_t)\right]^2 \tag{12.4.30}$$

$$\leq \mathbf{E}\left[f_a(Y_t)^2\right] + \mathbf{E}\left[f_a(Y_t)\right]^2 \tag{12.4.31}$$

First, we proceed to bound $\mathbf{E}\left[f_a(Y_t)^2\right]$. Consider a coupled execution of synchronous Gibbs sampling $X_0, X_1, \ldots, X_t$ where $X_0 = Y_0$ coupled using the greedy coupling. We have,

$$\mathbf{E}\left[f_a(Y_t)^2\right] \leq \mathbf{E}\left[f_a(X_t)^2\right] + \left|\mathbf{E}\left[f_a(X_t)^2 - f_a(Y_t)^2\right]\right| \tag{12.4.32}$$

$$\leq C_1(2d, \alpha)\left(n \log n\right)^d + C_2(2d, \alpha, \tau)\left(n \log n\right)^{(2d-1)/2} \leq C_3(d, \alpha, \tau)\left(n \log n\right)^d \tag{12.4.33}$$

where we used the fact that $f(x)^2$ is a degree $2d$ polynomial and then applied Theorem 12.4.4 for

degree $2d$ polynomials, in addition to Theorem 12.2.9. Now we look at the second term of (12.4.31).

$$\mathbf{E}\left[f_a(Y_t)\right]^2 \leq \mathbf{E}\left[f_a(X_t)\right]^2 + \left|\mathbf{E}\left[f_a(Y_t)\right]^2 - \mathbf{E}\left[f_a(X_t)\right]^2\right| \tag{12.4.34}$$

$$\leq \mathbf{E}\left[f_a(X_t)\right]^2 + \left|\left(\mathbf{E}\left[f_a(Y_t)\right] - \mathbf{E}\left[f_a(X_t)\right]\right)\left(\mathbf{E}\left[f_a(Y_t)\right] + \mathbf{E}\left[f_a(X_t)\right]\right)\right| \tag{12.4.35}$$

$$\leq C_1^2(d,\alpha)\left(n\log n\right)^d + C_2(d,\alpha,\tau)\left(n\log n\right)^{(d-1)/2}\left(\left|2\mathbf{E}\left[f_a(X_t)\right]\right| + \left|\mathbf{E}\left[f_a(Y_t)\right] - \mathbf{E}\left[f_a(X_t)\right]\right|\right) \tag{12.4.36}$$

$$\leq C_4(d,\alpha,\tau)\left(n\log n\right)^{(2d-1)/2}, \tag{12.4.37}$$

where again we employ Theorem 12.2.9 and Theorem 12.4.4 for degree d polynomials. Combining (12.4.33) and (12.4.37), we get the desired bound. □

12.4.1 Going Beyond Ising Models

We presented results for accurate estimation of polynomial functions over the Ising model. However, the results can be extended to hold for more general graphical models satisfying Dobrushin's condition. A main ingredient here was concentration of measure. If the class of functions we look at has d^{th}-order bounded differences in expectation, then we indeed get concentration of measure for these functions (Theorem 1.2 of [81]). This combined with the techniques in our work would allow similar gains in accurate estimation of such functions on general graphical models.

12.5 Experiments

We show the results of experiments run on a machine with four 10-core Intel Xeon E7-4850 CPUs to demonstrate the practical validity of our theory. In our experiments, we focused on two Ising models—Curie-Weiss and the Grid. The Curie-Weiss $CW(n,\alpha)$ is the Ising model corresponding to the complete graph on n vertices with edges of weight $\beta = \frac{\alpha}{n-1}$. The $Grid(k^2,\alpha)$ model is the Ising model corresponding to the k-by-k grid with the left connected to the right and top connected to the bottom to form a torus—a four-regular graph; the edge weights are $\frac{\alpha}{4}$. The total influence of each of these models is at most α, so we chose $\alpha = 0.5$ to ensure Dobrushin's condition. To generate samples, we start at a uniformly random configuration and run Markov chains for $T = 10n\log_2(n)$ steps to ensure mixing.

In our first experiment (Figure 12.5.1) we validate the modeling assumption that the average delay of a read τ is a constant. Computing the exact delays in a real run of the HOGWILD! is not

possible, but we approximate the delays by making processes log read and write operations to a lock-free queue as they execute the HOGWILD!-updates. We present two plots of the average delay of a read in a HOGWILD! run of the $CW(n, 0.5)$ Markov chain with respect to n. Four asynchronous processors were used to generate the first plot, while twenty were used for the second. We notice that the average delay depends on the number of asynchronous processes, but is constant with respect to n as assumed in our model.

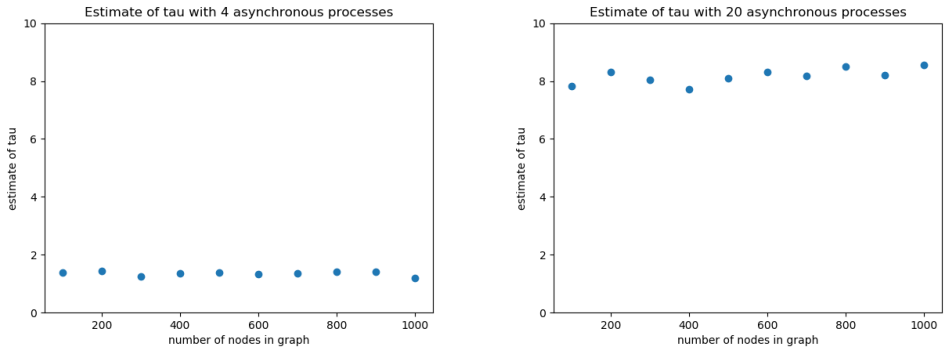


Figure 12.5.1: Average delay of reads for $CW(n, 0.5)$ model. Four asynchronous processors were used on the left, while twenty were used on the right.

Next, we plot (in Figure 12.5.2) the relationship between the number of asynchronous processors used in a HOGWILD! execution and the delay parameter τ. For this plot, we estimated τ by the average empirical delay over HOGWILD! runs of $CW(n, 0.5)$ models, with n ranging from 100 to 1000 in increments of one hundred. The plot shows a linear relationship, and suggests that the delay per additional processor is approximately 0.4 steps.

The primary purpose of our work is to demonstrate that polynomial statistics computed from samples of a HOGWILD! run of Gibbs Sampling will approximate those computed from a sequential run. Our third experiment demonstrates exactly this fact. We plot (in Figure 12.5.3 on the left) the empirical expectations of the *complete bilinear function* $f(X_1, \ldots, X_n) = \sum_{i \neq j} X_i X_j$ as we vary the number of nodes n in a Curie-Weiss model graph. Each red point is the empirical mean of the function f computed over 5000 samples from the HOGWILD! Markov chain corresponding to $CW(n, 0.5)$, and each blue point is the empirical mean produced from 5000 sequential runs of the same chain. Our theory (Theorem 12.4.3) predicts that the bias, the vertical difference in height between red and blue points, at any given value of n will be on the order of the standard deviation divided by $\sqrt{n}$ (standard deviation is $\Theta(n)$ and bias is $O(\sqrt{n})$). We plot error bars of this order, and find that the HOGWILD! means fall inside the error bars, thus corroborating our theory. We

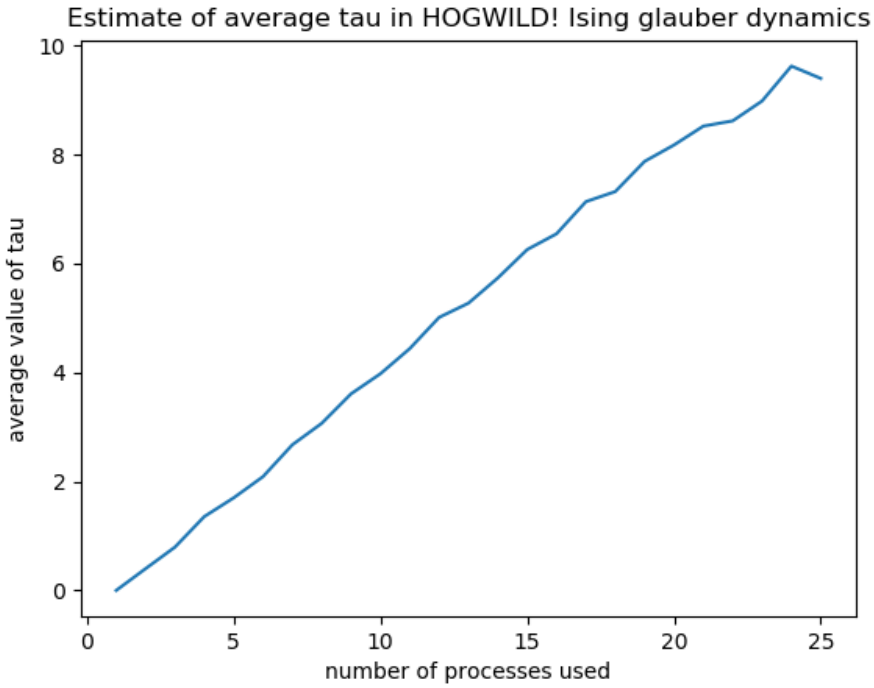


Figure 12.5.2: Average delay of reads for $CW(n, 0.5)$ model as the number of processors used varies.

show that theory and practice coincide even for sparse graphs, by making the same plot for the $Grid(n, 0.5)$ model on the right of the same figure.

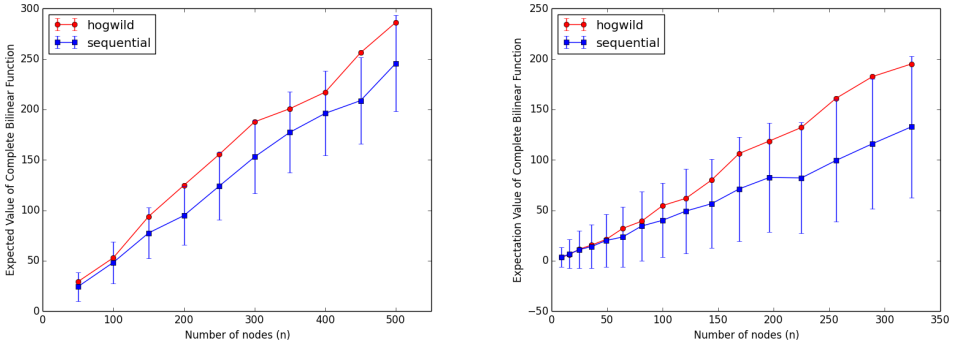


Figure 12.5.3: Means (with appropriately scaled error bars) of the complete bilinear function computed over 5000 sequential and hogwild runs of $CW(n, 0.5)$ (left) and $Grid(n, 0.5)$ (right).

Part VI

Conclusion

Chapter 13

Summary

In this thesis, I presented multiprocess algorithms for a range of problems spanning concurrency, data structures, parallel algorithms and machine learning. Throughout the process, I emphasized the themes of simplicity, speed, scalability, and reliability.

In Part II, we saw how the process of lock design started out with simplifying the queue lock to its core. This effort, presented in Chapter 4 resulted in the short, eight-line JJJ Lock algorithm, which is a most efficient lock in each of the CC and DSM models, unifies the node-switching and node-toggling methods, and supports the union of all desirable properties we know for queue locks, such as first-come-first-served and bounded exit. In particular, the JJJ Lock is retains all the key structures in past queue locks, while remaining minimalist. Thus, it set the foundation for the design of our abortable and recoverable locks. Our abortable locks are the most efficient ones known to date, with just constant remote memory reference (RMR) complexity per passage. They are also the first to achieve the coveted, *fast abort* property, of worst-case constant RMRs to abort. The simplicity of our queue lock structure was also key to our provably optimally efficient recoverable mutual exclusion locks.

Along with achieving efficiency, i.e. speed and scalability, our locks come with rigorous proofs of their correctness and other properties. Notably, we give induction proofs of the key invariants of *all* of our lock algorithms. While these invariant proofs may seem tedious at times, their importance is particularly well illustrated in the case of recoverable mutual exclusion, where we showed that a previous state-of-the-art algorithm, thought to be correct, actually has subtle bugs and races that sometimes result in starvation and, worse yet, deadlock. Particularly novel in our proof methods, is our proof of starvation freedom of our abortable mutex. To the best of our knowledge, we are

the first to introduce the *distance function* method for proving progress occurs in all fair runs of a mutual exclusion lock. The efficiency analysis for the same algorithm is somewhat novel in its use of a multiprocess potential function, a technique we also used in our union-find data structure efficiency analysis.

Moving past locks, in Part III, we investigated *lock-free* data structures, both from a perspective of efficient algorithms and from a perspective of fundamental lower bounds. Our main algorithmic results in this part were a family of fast algorithms for concurrent union-find, fast arrays, fast generalized arrays, and fixed size hash tables. In our analysis, we revealed the importance of accounting for possible *randomness leaks* when trying to prove any efficiency claims about a randomized concurrent algorithms. Randomness leaks, which result when the adversary is able to discover more about the algorithm's random bits through clever scheduling, are a key difficulty in getting good efficiency results in the asynchronous model. They have also been significantly overlooked in some previous analyses, including the analysis of some hash tables. The highlight of our algorithmic results, is our efficient wait-free concurrent union-find object, which has work complexity $O(\alpha(n) + \log p)$ even against the adaptive scheduling adversary. This algorithm is the first provably scalable algorithm for union-find, and indeed any common data structure we are aware of (other than Ellen and Woelfel's fetch-and-increment construction).

For arrays, we showed the surprising result that we can present the interface of initializing an entire m-length array in just $O(1)$ time, while still performing read, write, and indeed every other primitive operation supported by hardware (such as CAS, FAS, etc.) in just $O(1)$ time each. This result fascinated me at first, since it seems to counteract the powerful super-linear lower bounds that researchers have shown for the write-all problem—essentially the problem of initializing an array of length m to all zeros. In our work, we showed how to use these fast arrays to develop a provably efficient fixed-length hash table. I believe that this result maybe a hotbed of future algorithmic development, since arrays are so fundamental, and the problem of initializing arrays in the face of concurrency is an even more difficult task than initializing sequentially.

On the lower bounds side of our work, we showed that the tight upper bounds we prove for our family of union-find data structures are actually optimal among the class of *symmetric* data structures for union-find and that a less power $\Omega(\log \log p)$ lower bound holds on *all* data structures for union-find. This leaves a small gap, that a potentially faster non-symmetric algorithm or a stronger lower bound must eventually fill. Our conjecture is that the lower bound is the one that can be improved, and that the union-find algorithm is likely optimal. We also showed that many common

data structures such as stacks, queues, and priority queues cannot have concurrent linearizable implementations without a $\Omega(\log p)$ amortized work overhead of concurrency per operation. These lower bounds lead us to argue that the goal of future concurrent data structures should be to derive algorithms whose work complexities scale polylogarithmically in data set size n and process-count p.

Part IV contains our most significant work focused on improving the reliablility of multiprocessor algorithms. In particular, we identify that linearizability has been the gold standard for multiprocessor data structure correctness for decades, but proofs of linearizability continue to be hard to pharse, hard to verify, and thus many times lead to unproved or even wrong published algorithms. We mitigate this state of affairs by introducing a simple, universal, sound, and complete method for enabling *machine-verified* proofs of the linearizability, and even the strong linearizability, of any implementation (that is indeed linearizable or strong linearizable). As a crowning result, we prove the linearizability and strong linearizability of our union-find algorithm using our method, and have made the proof publicly available on GitHub to be independently verified by whomsoever wishes to do so. My collaborators and I have also designed publicly available machine-verified proofs of several other data structures in related work. I certainly hope that more and more multiprocessor algorithms will become provably reliable through this system of publicly available machine-verified proofs of correctness.

Finally, in Part V, we turned to an efficiency problem that traditionally belonged solely to machine learning—sampling high dimensional probability distributions, which are supplied as graphical models. In this part, we introduced the stochastic scheduling model of asynchronous computing, and under this model, whose validity we bolstered experimentally, we proved statistical guarantees of an extremely simple, race-condition heavy, lock-free multiprocessor-algorithm. The powerful results of this part, that show that HOGWILD! Gibbs Sampling can be *pan-accurate*, i.e., generate accurate statistics about functions of all variables of a graphical model, required an analysis that mixes traditional asynchronous computing techniques with concepts from probability theory, such as markov chains and filtering, and techniques from machine learning, such as Dobrushin's coefficient.

Along with its technical contributions, this thesis also pioneered a new direction in broadening the participation in STEM across linguistic communities. by presenting, to my knowledge, the first modern computer science research originally penned in Telugu, which as accompanied by an abstract in Sanskrit—which is aimed to reach an even larger number of people.

Chapter 14

Future Directions

Any thesis of this length naturally opens the doors to hundreds of open directions. Rather than make a hopeless attempt at listing all of these, I will put forth just a few future directions for technical work.

- Fast and Scalable Wait-Free Linearizable Data Structures: The principle goal of this thesis was to design, and aid in the design of, fast and scalable multiprocess algorithms. Design of such efficient algorithms is propelled by the design of efficient wait-free data structures. The lower bounds of Chapter 9 show that we simply cannot design linearizable stacks and queues without a concurrency overhead of at least $\Omega(\log p)$ per operation. I believe an important open direction is designing scalable lock-free linearizable data-structures for fundamental problems like stacks, queues, priority queues, and dictionaries. As I argued in this thesis, I believe the goal should be to achieve algorithms whose work complexity is at most polylogarithmic in both n, the number of items in the data structure, and p the total number of processes that use the shared data structure. As we demonstrated with union-find, such algorithms are likely to have far-reaching consequences in fast parallel algorithms for several problems of interest.

- Closing the Complexity Gap for Union-Find: It is well known that the analysis of the sequential union-find data structure [198] and the lower bounds work on the sequential union-find problem [64] were foundational in the fields of algorithmic analysis. I believe a deep probe into the computational limits of concurrent union-find can give us a similar deep understanding. I conjecture that the gap between our upper and lower bound will be resolved in favor of the algorithm, i.e., a better lower bounding technique is what is left to be discovered. Thus, I

believe this is an open direction that is mainly of theoretical interest.

- Machine Verification Techniques: The machine-verifiable proof techniques that we introduced in the work of Chapter 11 are, in my opinion, our biggest contribution towards *reliability* of multiprocessor algorithms. I believe that many of the proofs we have seen in this thesis apart from just the linearizability proofs—for instance, the proofs of correctness of the mutex lock invariants—can be made machine-verifiable. Perhaps more significantly, I want to move beyond simply focusing on correctness, but also focusing on efficiency. My future goal is to develop techniques and generate machine-verified proofs of efficiency properties of multiprocess algorithms—for instance, giving machine-verified proofs of results like the inverse-Ackermann efficiency of our concurrent union-find object or the amortized constant RMR complexity of our abortable locks.

Future directions towards broadening linguistic diversity in STEM

The productive grammar of Sanskrit [168] served as an incredibly powerful tool in deriving technical vocabulary to make mathematical and scientific research exposition possible in Telugu. In fact, I have found that deriving Sanskrit vocabulary for technical terms in scientific exposition comes with at least three significant, positive consequences:

1. Sanskrit is the progenitor of the technical vocabulary in innumerable languages across the world, including but not limited to most Indian and South Asian languages. Thus, terms derived in Sanskrit's grammar serve as a unifying vocabulary for billions of people. In the same way that an English speaker can largely understand a mathematical work in French without a deep background in the French language due to the etymological similarity between, for instance, the French *intégrale* and the English *integral*, speakers of Sanskrit-descendant languages can communicate across language barriers with shared Sanskrit vocabulary.

2. Sanskrit has an ancient yet still thriving tradition of formal grammar, expressed in an algorithmic form in Pānini's celebrated Ashtādhyāyī [168]. This grammar offers a unique opportunity for building a long lasting vocabulary, where the semantic meaning of technical words can drive their morphological form, thereby embedding these words in a framework that has lasted over two millennia. (On a personal note: as an algorithmist, I also find the aesthetic beauty of an algorithmic treatise driving not just the content, but also the form of scientific

terminology appealing.)

3. New scientific and technical vocabulary is currently *the* principal barrier to enabling the writing of scientific texts in many languages. Thus, developing Sanskrit vocabulary ameliorates this principal barrier for many language communities simultaneously.

Proposal: I thus propose the *Samskrtam Technical Lexicon Project*. As I envision it, the project should aim to use Pānini's productive grammar of Sanskrit to create a dictionary of modern technical terms along with their etymological derivations (व्युत्पत्ति, *vyutpatti*) from Sanskrit roots (dhātu), prefixes (upasarga), suffixes (pratyaya), and compounds (samāsa), and a description of how to use these words in the many languages—including for instance Telugu—that derive vocabulary from Sanskrit. To achieve the best possible outcomes, I propose forging a close collaboration between scholars of STEM fields and scholars of Sanskrit and other vernaculars to implement this vision.

A final note I am glad to have worked on the *TeluguTeX* template, built with XeLaTeX, to overcome the lack of software to easily typeset scientific text in Telugu. Telugu and various other languages continue to suffer from technological under-availability in several areas. To address this concern, I support making more rapid progress on existing technological initiatives that empower linguistic diversity, such as Unicode [18], XeLaTeX [128], and Internet Internationalization [99].

Bibliography

[1] Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. "Making Objects Writable." In: *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 2014. DOI: 10.1145/2611462.2611483.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Dan Alistarh, Alexander Fedorov, and Nikita Koval. "In Search of the Fastest Concurrent Union-Find Algorithm." In: *23rd International Conference on Principles of Distributed Systems, OPODIS 2019*. 2019. DOI: 10.4230/LIPIcs.OPODIS.2019.15.

[4] Dan Alistarh et al. "The SprayList: A Scalable Relaxed Priority Queue." In: *SIGPLAN Not.* (2015). DOI: 10.1145/2858788.2688523.

[5] Dan Alistarh et al. "The SprayList: A Scalable Relaxed Priority Queue." In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2015. DOI: 10.1145/2688500.2688523.

[6] Josh Alman and Virginia Vassilevska Williams. "A Refined Laser Method and Faster Matrix Multiplication." In: *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2021. DOI: 10.1137/1.9781611976465.32.

[7] Adam Alon and Adam Morrison. "Deterministic Abortable Mutual Exclusion with Sublogarithmic Adaptive RMR Complexity." In: *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing*. 2018.

[8] *Amazon EC2 High Memory Instances*. https://aws.amazon.com/ec2/instance-types/high-memory/. Accessed: November 1, 2022.

[9] Daphna Amit et al. “Comparison Under Abstraction for Verifying Linearizability.” In: *Computer Aided Verification CAV 2007*. Springer, 2007. DOI: 10.1007/978-3-540-73368-3_49.

[10] James H. Anderson and Yong-Jik Kim. “Local-spin Mutual Exclusion Using Fetch-and-\phi Primitives.” In: *International Conference on Distributed Computing Systems (ICDCS) 2003)*. 2003. DOI: 10.1109/ICDCS.2003.1203505.

[11] Richard J. Anderson and Heather Woll. “Algorithms for the Certified Write-All Problem.” In: *SIAM J. Comput.* 26.5 (1997), pp. 1277–1283.

[12] Richard J. Anderson and Heather Woll. “Wait-free Parallel Algorithms for the Union-Find Problem.” In: *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*. 1991. DOI: 10.1145/103418.103458.

[13] T. E. Anderson. “The performance of spin lock alternatives for shared-money multiprocessors.” In: *IEEE Transactions on Parallel and Distributed Systems* (1990). DOI: 10.1109/71.80120.

[14] Anish Athalye et al. “Synthesizing Robust Adversarial Examples.” In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2018.

[15] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. “Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory.” In: *PODC*. ACM, 2018. DOI: 10.1145/3212734.3212753.

[16] Hagit Attiya and Constantin Enea. “Putting Strong Linearizability in Context: Preserving Hyperproperties in Programsthat Use Concurrent Objects.” In: *International Symposium on Distributed Computing, DISC 2019*. Ed. by Jukka Suomela. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. DOI: 10.4230/LIPIcs.DISC.2019.2.

[17] Hagit Attiya, Danny Hendler, and Philipp Woelfel. “Tight RMR Lower Bounds for Mutual Exclusion and Other Problems.” In: *Proc. of the Fortieth ACM Symposium on Theory of Computing*. 2008.

[18] Joseph D Becker. *Unicode 88*. 1988.

[19] Jon Louis Bentley. *Programming pearls*. Addison-Wesley, 1986.

[20] Josh Berdine et al. “Thread Quantification for Concurrent Shape Analysis.” In: *Computer Aided Verification, (CAV)*. Springer, 2008. DOI: 10.1007/978-3-540-70545-1_37.

[21] Ananya Bhattacharya. *This is America's fastest growing language. Clue: It might not be what you expect.* 2018. URL: https://www.weforum.org/agenda/2018/10/america-s-fastest-growing-foreign-language-is-from-south-india.

[22] Vincent Bloemen. "On-The-Fly Parallel Decomposition of Strongly Connected Components." MA thesis. University of Twente, 2015.

[23] Vincent Bloemen. "Strong Connectivity and Shortest Paths for Checking Models." PhD thesis. University of Twente, 2019. DOI: 10.3990/1.9789036547864.

[24] Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. "Multi-Core on-the-Fly SCC Decomposition." In: Association for Computing Machinery, 2016. DOI: 10.1145/3016078.2851161.

[25] Norbert Blum. "On the single-operation worst-case time complexity of the disjoint set union problem." In: *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*. 1985.

[26] Robert D. Blumofe et al. "Cilk: An Efficient Multithreaded Runtime System." In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*. 1995. DOI: 10.1145/209936.209958.

[27] Enric Boix-Adserà, Benjamin L. Edelman, and Siddhartha Jayanti. "The Multiplayer Colonel Blotto Game." In: *EC '20: The 21st ACM Conference on Economics and Computation*. 2020. DOI: 10.1145/3391403.3399555.

[28] Enric Boix-Adserà, Benjamin L. Edelman, and Siddhartha Jayanti. "The Multiplayer Colonel Blotto Game." In: *Games and Economic Behavior* (2021). DOI: 10.1016/j.geb.2021.05.002.

[29] *Broadening Participation in STEM*. https://beta.nsf.gov/funding/initiatives/broadening-participation. 2022.

[30] Sebastian Burckhardt et al. "Line-up: a complete and automatic linearizability checker." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 2010. DOI: 10.1145/1806596.1806634.

[31] Jonathan F. Buss et al. "Parallel Algorithms with Processor Failures and Delays." In: *Journal of Algorithms* (1996).

[32] David Yu Cheng Chan and Philipp Woelfel. "Tight Lower Bound for the RMR Complexity of Recoverable Mutual Exclusion." In: (2021). DOI: 10.1145/3465084.3467938.

[33] Li Chen et al. "Maximum Flow and Minimum-Cost Flow in Almost-Linear Time." In: (2022). DOI: 10.48550/ARXIV.2203.00671.

[34] Benny Chor, Amos Israeli, and Ming Li. "On Processor Coordination Using Asynchronous Hardware." In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. Association for Computing Machinery, 1987. DOI: 10.1145/41840.41848.

[35] Fan Chung and Linyuan Lu. "Concentration Inequalities and Martingale Inequalities: A Survey." In: *Internet Mathematics Vol. 3, No. 1: 79-127*. Internet Mathematics, 2005. URL: http://people.math.sc.edu/lu/papers/concen.pdf.

[36] Richard Cole and Uzi Vishkin. "Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking." In: *Inf. Control* (1986). DOI: 10.1016/S0019-9958(86)80023-7.

[37] Robert Colvin and Lindsay Groves. "Formal Verification of an Array-Based Nonblocking Queue." In: *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China*. IEEE Computer Society, 2005. DOI: 10.1109/ICECCS.2005.49.

[38] Robert Colvin et al. "Formal Verification of a Lazy Concurrent List-Based Set Algorithm." In: *Computer Aided Verification, 18th International Conference, CAV*. Springer, 2006. DOI: 10.1007/11817963_44.

[39] D. Coppersmith and S. Winograd. "On the asymptotic complexity of matrix multiplication." In: *Annual Symposium on Foundations of Computer Science (SFCS)*. 1981. DOI: 10.1109/SFCS.1981.27.

[40] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[41] Travis S. Craig. *Building FIFO and Priority-Queuing Spin Locks from Atomic Swap*. Tech. rep. TR-93-02-02. Department of Computer Science, University of Washington, 1993.

[42] Robert Cypher. "The Communication Requirements of Mutual Exclusion." In: *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*. 1995. DOI: 10.1145/215399.215434.

[43] Yuval Dagan et al. “Learning from Weakly Dependent Data under Dobrushin’s Condition.” In: *Conference on Learning Theory, COLT 2019*. 2019.

[44] Constantinos Daskalakis, Nishanth Dikkala, and Siddhartha Jayanti. “HOGWILD!-Gibbs can be PanAccurate.” In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018*. 2018.

[45] Constantinos Daskalakis, Nishanth Dikkala, and Gautam Kamath. “Concentration of Multilinear Functions of the Ising Model with Applications to Network Data.” In: *Advances in Neural Information Processing Systems 30*. NIPS ’17. Curran Associates, Inc., 2017.

[46] Constantinos Daskalakis, Nishanth Dikkala, and Gautam Kamath. “Testing Ising Models.” In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2018.

[47] Constantinos Daskalakis, Elchanan Mossel, and Sébastien Roch. “Evolutionary Trees and the Ising Model on the Bethe Lattice: A Proof of Steel’s Conjecture.” In: *Probability Theory and Related Fields* (2011).

[48] Christopher De Sa, Kunle Olukotun, and Christopher Ré. “Ensuring rapid mixing and low bias for asynchronous Gibbs sampling.” In: *JMLR workshop and conference proceedings*. NIH Public Access. 2016.

[49] Christopher M De Sa et al. “Taming the wild: A unified analysis of hogwild-style algorithms.” In: *Advances in neural information processing systems*. 2015.

[50] R.H. Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions.” In: *Solid-State Circuits, IEEE Journal of* (1974). DOI: http://dx.doi.org/10.1023/A:1008373903657.

[51] Laxman Dhulipala, Changwan Hong, and Julian Shun. “ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms.” In: *Proc. VLDB Endow.* (2020). DOI: 10.14778/3436905.3436923.

[52] E. W. Dijkstra. “Solution of a Problem in Concurrent Programming Control.” In: *Communications of the ACM* (1965). DOI: 10.1145/365559.365617.

[53] Simon Doherty. “Modelling and verifying non-blocking algorithms that use dynamically allocated memory.” In: *Victoria University of Wellington*. 2003.

[54] Simon Doherty et al. “Formal Verification of a Practical Lock-Free Queue Algorithm.” In: *Formal Techniques for Networked and Distributed Systems - (FORTE)*. Springer, 2004. DOI: 10.1007/978-3-540-30232-2_7.

[55] Brijesh Dongol and John Derrick. “Verifying linearizability: A comparative survey.” In: *CoRR* abs/1410.6268 (2014). arXiv: 1410.6268. URL: http://arxiv.org/abs/1410.6268.

[56] R. Dvir and G. Taubenfeld. “Mutual exclusion algorithms with constant RMR complexity and wait-free exit code.” In: *International Conference on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. DOI: 10.4230/LIPIcs.OPODIS.2017.17.

[57] Faith Ellen and Philipp Woelfel. “An Optimal Implementation of Fetch-and-Increment.” In: *Proceedings of the International Symposium on Distributed Computing (DISC)*. 2013. DOI: 10.1007/978-3-642-41527-2_20.

[58] Glenn Ellison. “Learning, Local Interaction, and Coordination.” In: *Econometrica* (1993).

[59] Urban Engberg, Peter Grønning, and Leslie Lamport. “Mechanical Verification of Concurrent Systems with TLA.” In: *Proceedings of the International Workshop on Larch*. Springer, 1992.

[60] Panagiota Fatourou, Nikolaos D. Kallimanis, and Thomas Ropars. “An Efficient Wait-Free Resizable Hash Table.” In: *Proceedings of the on Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Association for Computing Machinery, 2018. DOI: 10.1145/3210377.3210408.

[61] Joseph Felsenstein. *Inferring Phylogenies*. Sinauer Associates Sunderland, 2004.

[62] Michael J Fischer. “Efficiency of equivalence algorithms.” In: *Complexity of Computer Computations*. Springer, 1972.

[63] Wojciech Fraczak et al. “Finding dominators via disjoint set union.” In: *Journal of Discrete Algorithms* (2013).

[64] Michael L. Fredman and Michael E. Saks. “The Cell Probe Complexity of Dynamic Data Structures.” In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC)*. 1989. DOI: 10.1145/73007.73040.

[65] Kimmo Fredriksson and Pekka Kilpeläinen. “Practically efficient array initialization.” In: *Softw. Pract. Exp.* (2016).

[66] Joel Friedman. "A proof of alon's second eigenvalue conjecture." In: *the Proceedings of the annual ACM symposium on Theory of computing (STOC)*. 2003.

[67] Bernard A Galler and Michael J Fisher. "An improved equivalence algorithm." In: *Communications of the ACM* (1964).

[68] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. "Almost Wait-Free Resizable Hashtable." In: *International Parallel and Distributed Processing Symposium*. 2004.

[69] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. "Lock-free dynamic hash tables with open addressing." In: *Distributed Computing* (2005). DOI: 10.1007/s00446-004-0115-2.

[70] Stuart Geman and Christine Graffigne. "Markov Random Field Image Models and their Applications to Computer Vision." In: *Proceedings of the International Congress of Mathematicians*. American Mathematical Society, 1986.

[71] Reza Gheissari, Eyal Lubetzky, and Yuval Peres. "Concentration inequalities for polynomials of contracting Ising models." In: *arXiv preprint arXiv:1706.00121* (2017).

[72] George Giakkoupis and Philipp Woelfel. "Randomized Abortable Mutual Exclusion with Constant Amortized RMR Complexity on the CC Model." In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2017. DOI: 10.1145/3087801.3087837.

[73] P. B. Gibbons. "A More Practical PRAM Model." In: *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*. 1989. DOI: 10.1145/72935.72953.

[74] Phillip B. Gibbons and Ephraim Korach. "Testing Shared Memories." In: *SIAM Journal on Computing* (1997). DOI: 10.1137/S0097539794279614.

[75] Ashish Goel et al. "Disjoint Set Union with Randomized Linking." In: *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2014. DOI: 10.1137/1.9781611973402.75.

[76] Wojciech Golab and Danny Hendler. "Recoverable Mutual Exclusion in Sub-logarithmic Time." In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2017.

[77] Wojciech Golab and Danny Hendler. "Recoverable Mutual Exclusion Under System-Wide Failures." In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 2018.

[78] Wojciech Golab and Aditya Ramaraju. “Recoverable Mutual Exclusion: [Extended Abstract].” In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. 2016.

[79] Wojciech Golab et al. “Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations.” In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. 2007. DOI: 10.1145/1281100.1281105.

[80] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. “Linearizable implementations do not suffice for randomized distributed computation.” In: *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. ACM, 2011. DOI: 10.1145/1993636.1993687.

[81] Friedrich Götze, Holger Sambale, and Arthur Sinulis. “Higher order concentration for functions of weakly dependent random variables.” In: *arXiv preprint arXiv:1801.06348* (2018).

[82] Ministry of Human Resource Development Government of India. “National Education Policy 2020.” In: (2020).

[83] G. Graunke and S. Thakkar. “Synchronization algorithms for shared-memory multiprocessors.” In: *IEEE Computers* (1990). DOI: 10.1109/2.55501.

[84] Jan Groote, Wim Hesselink, and Sjouke Mauw. “An Algorithm for the Asynchronous Write-All problem based on process collision.” In: *Distributed Computing* 14 (Apr. 2001), pp. 75–81.

[85] Torben Hagerup and Frank Kammer. “On-the-Fly Array Initialization in Less Space.” In: *International Symposium on Algorithms and Computation*. 2017.

[86] Shay Halperin and Uri Zwick. “Optimal Randomized EREW PRAM Algorithms for Finding Spanning Forests.” In: *Journal of Algorithms* (2001). DOI: https://doi.org/10.1006/jagm.2000.1146.

[87] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. “A Practical Multi-word Compare-and-Swap Operation.” In: *Proceedings of the 16th International Conference on Distributed Computing*. Springer-Verlag, 2002.

[88] Maurice Herlihy. “The Multicore Revolution.” In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. 2007.

[89] Maurice Herlihy. “Wait-free Synchronization.” In: *ACM Trans. Program. Lang. Syst.* (1991).

[90] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-free Data Structures." In: *SIGARCH Comput. Archit. News* (1993). DOI: 10.1145/173682.165164.

[91] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123705916.

[92] Maurice Herlihy and Jeannette M. Wing. "Axioms for Concurrent Objects." In: *Conference Record of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1987. DOI: 10.1145/41625.41627.

[93] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects." In: *ACM Trans. Program. Lang. Syst.* (1990).

[94] Lizzie Hernandez. "Formal Verification of a Snapshot Algorithm." Department of Computer Science: Dartmouth College, May 2023.

[95] Changwan Hong, Laxman Dhulipala, and Julian Shun. "Exploring the Design Space of Static and Incremental Graph Connectivity Algorithms on GPUs." In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (2020). DOI: 10.1145/3410463.3414657.

[96] Shlomo Hoory, Nathan Linial, and Avi Wigderson. "Expander graphs and their applications." In: *Bulletin of the American Mathematical Society* (2006).

[97] J. E. Hopcroft and J. D. Ullman. "Set Merging Algorithms." In: *SIAM Journal on Computing* (1973). DOI: 10.1137/0202024.

[98] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals*. 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[99] *Internationalized Domain Names for Applications (IDNA)*. 2003.

[100] Masakazu Ishihata, Shan Gao, and Shin-ichi Minato. "Fast Message Passing Algorithm Using ZDD-Based Local Structure Compilation." In: *Proceedings of the Workshop on Advanced Methodologies for Bayesian Networks*. 2017.

[101] Joseph F. JaJa. "PRAM (Parallel Random Access Machines)." In: *Encyclopedia of Parallel Computing*. 2011. DOI: 10.1007/978-0-387-09766-4_23.

[102] Prasad Jayanti. "A Complete and Constant Time Wait-Free Implementation of CAS from LL/SC and Vice Versa." In: *International Symposium on Distributed Computing (DISC)*. 1998. DOI: 10.1007/BFb0056485.

[103] Prasad Jayanti. "A Time Complexity Lower Bound for Randomized Implementations of Some Shared Objects." In: *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 1998. DOI: 10.1145/277697.277735.

[104] Prasad Jayanti. "Adaptive and Efficient Abortable Mutual Exclusion." In: *Proceedings of the Annual Symposium on Principles of Distributed Computing (PODC)*. 2003. DOI: 10.1145/872035.872079.

[105] Prasad Jayanti. "An optimal multi-writer snapshot algorithm." In: *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 2005. DOI: 10.1145/1060590.1060697.

[106] Prasad Jayanti. "F-arrays: Implementation and Applications." In: *Proceedings of the Annual Symposium on Principles of Distributed Computing (PODC)*. 2002. DOI: 10.1145/571825.571875.

[107] Prasad Jayanti and Siddhartha Jayanti. "Deterministic Constant-Amortized-RMR Abortable Mutex for CC and DSM." In: 2021. DOI: 10.1145/3490559.

[108] Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. "Towards an Ideal Queue Lock." In: *International Conference on Distributed Computing and Networking (ICDCN)*. 2020. DOI: 10.1145/3369740.3369784.

[109] Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. "Optimal Recoverable Mutual Exclusion using only FASAS." In: *The 6th Edition of The International Conference on Networked Systems*. Springer, Cham, 2018.

[110] Prasad Jayanti, Siddhartha V. Jayanti, and Anup Joshi. "A Recoverable Mutex Algorithm with Sub-logarithmic RMR on Both CC and DSM." In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 2019. DOI: 10.1145/3293611.3331634.

[111] Prasad Jayanti, Siddhartha V. Jayanti, and Anup Joshi. "Optimal Recoverable Mutual Exclusion Using only FASAS." In: *International Conference on Networked Systems (NETYS)*. 2018. DOI: 10.1007/978-3-030-05529-5_13.

[112] Prasad Jayanti and Anup Joshi. "Recoverable FCFS mutual exclusion with wait-free recovery." In: *International Symposium on Distributed Computing (DISC)*. 2017.

[113] Prasad V. Jayanti and Siddhartha Jayanti. "Constant Amortized RMR Abortable Mutex for CC and DSM." In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019. DOI: 10.1145/3293611.3331592.

[114] Siddhartha Jayanti. "Nash Equilibria of The Multiplayer Colonel Blotto Game on Arbitrary Measure Spaces." In: *CoRR* (2021).

[115] Siddhartha Jayanti, Srinivasan Raghuraman, and Nikhil Vyas. "Efficient Constructions for Almost-Everywhere Secure Computation." In: *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 2020. DOI: 10.1007/978-3-030-45724-2_6.

[116] Siddhartha Jayanti and Julian Shun. "Fast Arrays: Atomic Arrays with Constant Time Initialization." In: *35th International Symposium on Distributed Computing, DISC 2021*. 2021. DOI: 10.4230/LIPIcs.DISC.2021.25.

[117] Siddhartha Jayanti, Robert E. Tarjan, and Enric Boix-Adserà. "Randomized Concurrent Set Union and Generalized Wake-Up." In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 2019. DOI: 10.1145/3293611.3331593.

[118] Siddhartha V. Jayanti and Robert E. Tarjan. "A Randomized Concurrent Algorithm for Disjoint Set Union." In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2016. DOI: 10.1145/2933057.2933108.

[119] Siddhartha V. Jayanti and Robert E. Tarjan. "Concurrent disjoint set union." In: *Distributed Computing* (2021). DOI: 10.1007/s00446-020-00388-x.

[120] Siddhartha Visveswara Jayanti. "Generalized Wake-Up: Amortized Shared Memory Lower Bounds for Linearizable Data Structures." In: (). URL: https://arxiv.org/abs/2207.07561.

[121] Donald B Johnson and Panagiotis Metaxas. "Connected Components inO (log3/2 n) Parallel Time for the CREW PRAM." In: *journal of computer and system sciences* (1997).

[122] Matthew Johnson, James Saunderson, and Alan Willsky. "Analyzing hogwild parallel gaussian gibbs sampling." In: *Advances in Neural Information Processing Systems*. 2013.

[123] Mike Jones. *What really happened on Mars Rover Pathfinder*. cs.cornell.edu. 1997.

[124] Mike Jones. *What really happened to the software on the Mars Pathfinder spacecraft?* https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft. 2013.

[125] Paris C. Kanellakis and Alex A. Shvartsman. "Efficient Parallel Algorithms Can Be Made Robust." In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 1989.

[126] K Kasturirangan et al. *Draft National Education Policy 2019*. 2019.

[127] Takashi Katoh and Keisuke Goto. "In-Place Initializable Arrays." In: *CoRR* abs/1709.08900 (2017).

[128] Jonathan Kew. *XeTeX - Unicode-based TeX Code*. 2018.

[129] V. King. *A.M. Turing Award: Robert (Bob) Endre Tarjan with John E Hopcroft, for fundamental achievements in the design and analysis of algorithms and data structures*. URL: https://amturing.acm.org/award_winners/tarjan_1092048.cfm.

[130] Erica Klarreich. *Researchers Achieve 'Absurdly Fast' Algorithm for Network Flow*. 2022.

[131] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.

[132] Dexter C. Kozen. *The Design and Analysis of Algorithms*. Berlin, Heidelberg: Springer-Verlag, 1992.

[133] Rouven Krebs, Christof Momm, and Samuel Kounev. "Architectural Concerns in Multi-tenant SaaS Applications." In: *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*. 2012.

[134] Joseph B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." In: *Proceedings of the American Mathematical Society* (1956).

[135] Leslie Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem." In: *Commun. ACM* (1974). DOI: 10.1145/361082.361093.

[136] Chris Lattner and Vikram Adve. "Automatic Pool Allocation for Disjoint Data Structures." In: *SIGPLAN Not.* (2002). DOI: 10.1145/773039.773041.

[137] Hyonho Lee. "Fast Local-Spin Abortable Mutual Exclusion with Bounded Space." In: *Principles of Distributed Systems OPODIS*. 2010. DOI: 10.1007/978-3-642-17653-1_27.

[138] Hyonho Lee. *Local-spin mutual exclusion algorithms on the DSM model using fetch & store objects [microform]*. 2018.

[139] Charles Leiserson and Ilya Mirman. "How to Survive the Multicore Revolution." In: Cilk Arts, Inc, 2008.

[140] N.G. Leveson and C.S. Turner. "An investigation of the Therac-25 accidents." In: *Computer* (1993). DOI: 10.1109/MC.1993.274940.

[141] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2009.

[142] Joanne Lim. *An Engineering Disaster: Therac-25*. 1998.

[143] Ji Liu et al. "An asynchronous parallel stochastic coordinate descent algorithm." In: *The Journal of Machine Learning Research* (2015).

[144] Sixue Liu and Robert E. Tarjan. "Simple Concurrent Labeling Algorithms for Connected Components." In: *2nd Symposium on Simplicity in Algorithms (SOSA 2019)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. DOI: 10.4230/OASIcs.SOSA.2019.3.

[145] Yang Liu et al. "Model Checking Linearizability via Refinement." In: *Formal Methods (FM)*. Springer, 2009. DOI: 10.1007/978-3-642-05089-3_21.

[146] Jacob Teo Por Loong, Jelani Nelson, and Huacheng Yu. "Fillable arrays with constant time operations and a single bit of redundancy." In: *CoRR* abs/1709.09574 (2017).

[147] P. Magnusson, A. Landin, and E. Hagersten. *Queue locks on cache coherent multiprocessors*. Tech. rep. Swedish Institute of Computer Science, 1994. DOI: 10.1109/IPPS.1994.288305.

[148] Peter S. Magnusson, Anders Landin, and Erik Hagersten. "Queue Locks on Cache Coherent Multiprocessors." In: *Proceedings of the 8th International Symposium on Parallel Processing*. 1994. DOI: 10.1109/IPPS.1994.288305.

[149] Horia Mania et al. "Perturbed iterate analysis for asynchronous stochastic optimization." In: *arXiv preprint arXiv:1507.06970* (2015).

[150] C. Martel, R. Subramonian, and A. Part. "Asynchronous PRAMs are (almost) as good as synchronous PRAMs." In: *Proceedings of the IEEE Symposium on Foundations of Computer Science*. 1990.

[151] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Vol. 1. EATCS Monographs on Theoretical Computer Science. Springer, 1984.

[152] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for Scalable Synchronization on Shared-memory Multiprocessors." In: *ACM Trans. Comput. Syst.* (1991).

[153] Stephan Merz. "Proofs and Proof Certification in the TLA$^+$ Proof System." In: *Proceedings of the International Workshop on Proof Exchange for Theorem Proving (PxTP)*. CEUR-WS.org, 2012.

[154] M. M. Michael. "Hazard pointers: safe memory reclamation for lock-free objects." In: *IEEE Transactions on Parallel and Distributed Systems* (2004).

[155] Shin-ichi Minato. "Counting by ZDD." In: *Encyclopedia of Algorithms*. Springer Publishing Company, 2016.

[156] Shin-ichi Minato. "Power of Enumeration—Recent Topics on BDD/ZDD-Based Techniques for Discrete Structure Manipulation." In: *IEICE Trans. Inf. Syst.* (2017).

[157] Shin-ichi Minato. "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems." In: *Proceedings of the International Design Automation Conference*. 1993.

[158] Ioannis Mitliagkas et al. "FrogWild!: fast PageRank approximations on graph engines." In: *Proceedings of the VLDB Endowment* (2015).

[159] Andrea Montanari and Amin Saberi. "The Spread of Innovations in Social Networks." In: *Proceedings of the National Academy of Sciences* (2010).

[160] Gordon E. Moore. "Cramming more components onto integrated circuits." In: *Electronics* (1965).

[161] Dana Moshkovitz and Bruce Tidor. *Lecture Notes 15: van Emde Boas Data Structure*. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec15.pdf. May 2021.

[162] Motorolla. *Programmer's Reference Manual*. https://www.nxp.com/files-static/archives/doc/ref_manual/M68000PRM.pdf. Accessed: 2018-09-07.

[163] Gonzalo Navarro. "Constant-time array initialization in little space." In: *Proceedings of the International Conference of the Chilean Computer Science Society (SCCC)*. 2012.

[164] Gonzalo Navarro. "Spaces, Trees, and Colors: The algorithmic landscape of document retrieval on sequences." In: *ACM Comput. Surv.* (2013).

[165] Feng Niu et al. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent." In: *Advances in neural information processing systems*. 2011.

[166] Cyprien Noel and Simon Osindero. "Dogwild!-Distributed Hogwild for CPU & GPU." In: *NIPS Workshop on Distributed Machine Learning and Matrix Computations*. 2014.

[167] Linus Nyman and Mikael Laakso. "Notes on the History of Fork and Join." In: *IEEE Annals of the History of Computing* (2016). DOI: 10.1109/MAHC.2016.34.

[168] Pānini. *Ashtādhyāyī*. Around the 6th Century BCE.

[169] Abhijeet Pareek and Philipp Woelfel. "RMR-Efficient Randomized Abortable Mutual Exclusion." In: *CoRR* (2012). URL: http://arxiv.org/abs/1208.1723.

[170] Kevin Poulsen. "Software Bug Contributed to Blackout." In: (2004).

[171] Aditya Ramaraju. "RGLock: Recoverable mutual exclusion for non-volatile main memory systems." MA thesis. University of Waterloo, 2015. URL: https://uwspace.uwaterloo.ca/handle/10012/9473.

[172] Simone Raoux et al. "Phase-change random access memory: A scalable technology." In: *IBM Journal of Research and Development* (2008).

[173] Vibhor Rastogi et al. "Finding Connected Components on Map-reduce in Logarithmic Rounds." In: *Proceedings - International Conference on Data Engineering* (Mar. 2012). DOI: 10.1109/ICDE.2013.6544813.

[174] M. Raynal and D. Beeson. *Algorithms for Mutual Exclusion*. MIT Press, 1986.

[175] John Reif. "Depth First Search is Inherently Sequential." In: *Information Processing Letters 20* (1985).

[176] I. Rhee. "Optimizing a FIFO, scalable spin lock using consistent memory." In: *17th IEEE Real-Time Systems Symposium*. 1996. DOI: 10.1109/REAL.1996.563705.

[177] Siddhartha Sahu et al. "The ubiquity of large graphs and surprising challenges of graph processing: extended survey." In: *VLDB J.* (2020). DOI: 10.1007/s00778-019-00548-x.

[178] Gerhard Schellhorn, John Derrick, and Heike Wehrheim. "A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures." In: (2014). DOI: 10.1145/2629496.

[179] Michael L. Scott. "Non-blocking timeout in scalable queue-based spin locks." In: *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002*. 2002. DOI: 10.1145/571825.571830.

[180] Michael L. Scott and William N. Scherer III. "Scalable queue-based spin locks with timeout." In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. DOI: 10.1145/379539.379566.

[181] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[182] Ori Shalev and Nir Shavit. "Split-ordered lists: Lock-free extensible hash tables." In: *Journal of the ACM* (2006).

[183] M. Sharir. "A strong-connectivity algorithm and its applications in data flow analysis." In: *Computers & Mathematics with Applications* (1981). DOI: https://doi.org/10.1016/0898-1221(81)90008-0.

[184] Yossi Shiloach and Uzi Vishkin. "An O(logn) parallel connectivity algorithm." In: *Journal of Algorithms* (1982). DOI: https://doi.org/10.1016/0196-6774(82)90008-6.

[185] Julian Shun. *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*. Association for Computing Machinery and Morgan & Claypool, 2017.

[186] Alexander Smola and Shravan Narayanamurthy. "An architecture for parallel topic models." In: *Proceedings of the VLDB Endowment* (2010).

[187] Alexander J. Smola and Shravan M. Narayanamurthy. "An Architecture for Parallel Topic Models." In: *Proc. VLDB Endow.* (2010). DOI: 10.14778/1920841.1920931.

[188] Dmitri B Strukov et al. "The missing memristor found." In: *nature* (2008).

[189] "Summary by language size." In: (2022). URL: https://www.ethnologue.com/statistics/size.

[190] *Supermicro 8-Socket Intel Xeon 7U Rack Server*. https://happyware.com/uk-en/supermicro/sys-7089p-tr4t. Accessed: August 1, 2022.

[191] Hirofumi Suzuki and Shin-ichi Minato. "Fast Enumeration of All Pareto-Optimal Solutions for 0-1 Multi-Objective Knapsack Problems Using ZDDs." In: *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* (2018).

[192] Robert Tarjan. "Depth first search and linear graph algorithms." In: *Siam Journal on Computing* 1.2 (1972).

[193] Robert Tarjan. "Finding dominators in directed graphs." In: *SIAM Journal on Computing* 3.1 (1974), pp. 62–89.

[194] Robert Tarjan. “Testing flow graph reducibility.” In: *Proceedings of the annual ACM symposium on Theory of computing (STOC)*. 1973.

[195] Robert E. Tarjan. “An efficient planarity algorithm.” PhD thesis. Stanford University, USA, 1971.

[196] Robert E. Tarjan and Jan van Leeuwen. “Worst-case Analysis of Set Union Algorithms.” In: *J. ACM* (1984). DOI: 10.1145/62.2160.

[197] Robert Endre Tarjan. “A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets.” In: *J. Comput. Syst. Sci.* (1979). DOI: 10.1016/0022-0000(79)90042-4.

[198] Robert Endre Tarjan. “Efficiency of a Good But Not Linear Set Union Algorithm.” In: *J. ACM* (1975). DOI: 10.1145/321879.321884.

[199] Reality Check Team and BBC Telugu. “Do you speak Telugu? Welcome to America.” In: (2018). URL: https://www.bbc.com/news/world-45902204.

[200] Saied Tehrani et al. “Magnetoresistive random access memory using magnetic tunnel junctions.” In: *Proceedings of the IEEE* (2003).

[201] ఆప్తవక్త. మాణ్డూక్యోపనిషత్. ధర్మపరంపర, అనాది.

[202] Alexander Terenin, Daniel Simpson, and David Draper. “Asynchronous Gibbs Sampling.” In: *arXiv preprint arXiv:1509.08999* (2015).

[203] Florian Tramer et al. “On adaptive attacks to adversarial example defenses.” In: *Advances in Neural Information Processing Systems* (2020).

[204] Viktor Vafeiadis. “Automatically Proving Linearizability.” In: *International Conference on Computer Aided Verification (CAV)*. Springer, 2010. DOI: 10.1007/978-3-642-14295-6_40.

[205] Viktor Vafeiadis. “Shape-Value Abstraction for Verifying Linearizability.” In: *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 2009. DOI: 10.1007/978-3-540-93900-9_27.

[206] Viktor Vafeiadis et al. “Proving Correctness of Highly-Concurrent Linearisable Objects.” In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, 2006. DOI: 10.1145/1122971.1122992.

[207] Martin Vechev and Eran Yahav. “Deriving linearizable fine-grained concurrent objects.” In: 2008. DOI: 10.1145/1379022.1375598.

[208] Yiqiu Wang, Yan Gu, and Julian Shun. "Theoretically-Efficient and Practical Parallel DBSCAN." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*. 2020. DOI: 10.1145/3318464.3380582.

[209] Pierre Wolper. "Lectures on Formal Methods and Performance Analysis." In: *Lectures on Formal Methods and Performance Analysis*. Springer-Verlag New York, Inc., 2002. Chap. Constructing Automata from Temporal Logic Formulas: A Tutorial.

[210] Ugur Yavuz. "A Machine-Verified Proof of Linearizability for a Queue Algorithm." Code available at: https://github.com/uguryavuz/easy-to-verify-linearizability. Department of Computer Science: Dartmouth College, May 2022.

[211] Ugur Yavuz. "Easy-to-verify-linearizability." Department of Computer Science: Dartmouth College, June 2021.

[212] Ryo Yoshinaka et al. "Finding All Solutions and Instances of Numberlink and Slitherlink by ZDDs." In: *Algorithms* (2012).

[213] Hsiang-Fu Yu et al. "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems." In: *International Conference on Data Mining (ICDM)*. IEEE, 2012.

[214] Ce Zhang and Christopher Ré. "Dimmwitted: A study of main-memory statistical analytics." In: *Proceedings of the VLDB Endowment* (2014).

[215] సిద్ధార్థ విశ్వేశ్వర జయంతి. సామాన్య జాగృతిపరిష్కారం. ఆర్కైవ్, ౨౦౨౨.