Programming Technologies for Engineering Quality Multicore Software

by

Tim Kaler

B.S., Massachusetts Institute of Technology (2012)
M.Eng, Massachusetts Institute of Technology (2013)

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

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author . . . .
Department of Electrical Engineering and Computer Science
August 28, 2020

Certified by . . . .
Charles E. Leiserson
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

Programming Technologies for Engineering Quality Multicore Software

by

Tim Kaler

Submitted to the Department of Electrical Engineering and Computer Science on August 28, 2020, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Abstract

The widespread availability of large multicore computers in the cloud has given engineers and scientists unprecedented access to large computing platforms. Traditionally, high-end computing solutions have been developed and used by only a small community, as these solutions rely on expensive and specialized computing environments. The emergence of large-scale cloud computing providers, however, has democratized access to large-scale (although not necessarily HPC-scale) computing power, which can now be rented on-demand with just a credit card.

The complexity of parallel programming, however, has made it more difficult for even expert programmers to develop high-quality multicore software systems. For average programmers, developing parallel programs that are debuggable, correct, and performant is a daunting challenge. This thesis is concerned with the development of programming technologies that reduce the complexity of parallel programming to make it easier for average programmers to exploit the capabilities of multicore hardware.

I contend that realizing the full potential of the multicore revolution requires the development of programming technologies that make it easier to write ***quality code*** — code that has a simple understandable structure and performs well in practice. These programming technologies broadly include parallel algorithms, data structures, optimization techniques, profiling tools, and system design principles.

Along these ends, this thesis presents seven intellectual artifacts from the domains of parallel algorithms, multicore-centric systems for scientific computing, and programming tools that make it easier to write quality code by simplifying the design, analysis, and performance engineering of multicore software:

- **Chromatic**: Parallel algorithms for scheduling data-graph computations deterministically.
- **Color**: Parallel algorithms and ordering heuristics for graph coloring that have the simple semantics of serial code.
- **PARAD**: An efficient and parallelism-preserving algorithm for performing automatic differentiation in parallel programs.
- **Connectomics**: An end-to-end image-segmentation pipeline for connectomics using a single large multicore.
- **Alignment**: An image-alignment pipeline for connectomics that uses memory-efficient algorithms, and techniques for judiciously exploiting performance–accuracy tradeoffs.
- **Reissue**: Reissue policies for reducing tail-latency in distributed services that are easy to analyze and effective in practice.
- **Cilkmem**: Efficient algorithms and tools for measuring the worst-case memory high-water mark of parallel programs.

Although the emphasis and domains of these artifacts vary, they each involve the discovery of a way to tame complexity in parallel software systems without compromising, in fact, usually enhancing, theoretical guarantees and real-world performance.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

There are many individuals to whom I owe my thanks for their support and guidance during my graduate studies.

First and foremost, I thank my advisor Charles E. Leiserson. Charles has earned my enduring gratitude for his invaluable support, guidance, and generosity over the years. Charles is a professor who exudes integrity, curiosity, and a sense of fun. These attributes motivated me to work hard in his undergraduate algorithms class, take his performance engineering class, and ultimately join his research group. Since then I've learned more about writing and research from Charles than I could have ever anticipated. I can't say that it has always been easy to implement all of his advice, but I can say that it's always been worth the effort.

I thank Tao B. Schardl, Julian Shun, and I-Ting Angelina Lee for their service on my thesis committee and for helpful discussions over the years. Tao Schardl has a deep understanding of parallel linguistics, performance engineering, and compilers and has been a constant source of ideas, inspiration, and support. I also thank Tao Schardl for his contributions to the Chromatic, Color, Cilkmem, and PARAD papers. Julian has shared his substantial expertise in the design of work-efficient parallel algorithms and related topics. I also thank Julian for inviting me to give a lecture in his seminar class. Angelina has provided insights into the internals of the Cilk runtime system, and helpful discussions on research directions on many topics including reducer hyperobjects, scheduling, and program analysis tools.

Nir Shavit deserves thanks for his support, encouragement, and contributions to the Connectomics and Alignment papers. The field of connectomics has fascinated me since I was in high school, and it was a joy to contribute to the project. I thank all the members of the connectomics project, especially Alex Matveev and Yaron Meirovitch, with whom I've had many fruitful and enriching discussions. I also thank my collaborators at Harvard including Daniel Berger, Adi (Suissa) Peleg, Thouis R. Jones, Hanspeter Pfister, and Jeff Lichtman.

Yuxiong He and Sameh Elnikety provided valuable mentorship during my time at Microsoft Research and contributed to the Reissue paper. Their good humor and sharp intellectual insights made my time at MSR an absolute joy.

I thank my collaborators at IBM, especially Jie Chen, Georgios Kollias and Mark Weber, for hosting me during multiple visits to IBM's campus and for advice and helpful discussions related to machine learning on graphs and automatic differentiation.

I thank my other coauthors, not yet mentioned, William Kuszmaul, Daniele Vettorel, William Hasenplaugh, Brian Wheatman, Sarah Wooders, Erik Demaine, Quanquan Liu, Adam Yedidia, and Aaron Sidford. I thank William Kuszmaul for his unique theoretical insights into approximation algorithms in the Cilkmem paper. I thank Daniele Vetorrel for his contributions to the Cilkmem paper, and for his advice and guidance in designing a prototype CSI tool for the PARAD paper. I thank Brian Wheatman and Sarah Wooders for their dedication and persistence as undergraduate researchers and M.Eng students during our collaboration on the Alignment paper. I thank Erik Demaine, Quanquan Liu, Adam Yedidia, and Aaron Sidford for their theoretical insights and contributions to the Retroactive paper. I thank William Hasenplaugh for his contributions to the Chromatic and Color papers as well as for his good humor and for sharing his deep theoretical and practical understanding of memory models and computer architecture.

Julian Shun and Adam Belay served on my RQE commmittee and I am grateful for their helpful comments and advice.

I thank all of the students, faculty, and staff on the seventh floor for providing helpful discussions and fun diversions throughout my graduate studies. I thank the members of the Supertech research group past and present. I especially thank Bradley Kuszmaul, Maryam Mehri Dehnavi, Shahin Kamali, Helen Xu, and Matthew Kilgore for their unique insights and helpful discussions over the years. I give special thanks to Cree Bruins and Marsha Davidson for their logistical support and for adding an extra dose of levity and warmth to the atmosphere of the group.

The members of the EECS graduate office deserve my thanks for helping my graduate studies run smoothly. I especially thank Janet Fischer for her assistance over the years. I thank Anne Hunter and the members of the undergraduate EECS office for all of the support provided during my undergraduate studies, and for their support of my UROP and M.Eng students.

The members of TIG have been consistently helpful and flexible while I've been at CSAIL. The dedication of the members of TIG to maintaining transparency and openness in the network and computing infrastructure is something I really appreciate. Garrett Wollman, in particular, has been especially helpful for various computing projects related to courses and research.

I thank all of the faculty, staff, and students that have provided support during my time as an undergraduate at MIT. In particular, I thank Hari Balakrishnan, Sam Madden, Ben Waber, Lenin Ravindranath, and Arvind Thiagarajan for giving me early opportunities to engage in research during my undergraduate studies. I owe special thanks to Jeremy Orloff and Gabrielle Stoy for their excellent mentorship, infectious appreciation for good mathematical arguments, and giving me the opportunity to learn to teach during my first few years as an undergraduate at MIT.

I am tremendously grateful for the support of my research sponsors. This research was sponsored in part by: the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000; the National Science Foundation (NSF) under grants IIS-1447786, CNS-1017058, CCF-1162148, CCF-1314547, CCF-1563880; the Intelligence Advanced Research Projects Activity (IARPA) under grant 138076-5093555; the MIT-IBM Watson AI Lab under grant 027397-00059; the Intel Corporation; and, Foxconn Technology Group. The views and conclusions contained in this document are mine and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Last, but not least, I thank all friends and family that have lent their support during my graduate studies. I especially thank Sophia for her support and encouragement during my graduate studies and during the final stretch of writing my thesis.

Contents

Chapter 1

Introduction

The widespread availability of large multicore hardware in the cloud has given engineers and scientists unprecedented access to large computing platforms. Traditionally, high-end computing solutions have been developed and used by only a small community, as these solutions rely on expensive and specialized computing environments. The emergence of large-scale cloud computing providers, however, has democratized access to large-scale (although not necessarily HPC-scale) computing power, which can now be rented on-demand with just a credit card.

Modern multicore hardware can store within their primary memory the datasets of many big research problems in engineering and science. For example, the hyperlink graph extracted from the Common Crawl 2012 web corpus, the largest publicly available graph of the World Wide Web [257, 256], has fewer than 3.6 billion vertices and 129 billion edges and can be stored in much less than 100GB in a variety of formats. The largest graph in Stanford's SNAP Datasets [222] is com-Friendster, which has fewer than 66 million vertices and just over 1.8 billion edges and can be stored in less than 9GB. The largest sparse matrix currently in the University of Florida's Sparse Matrix Collection [84] is sk-2005, which has just over 50 million rows and columns and fewer than 2 billion nonzeros and can be stored in less than 3GB. All of these datasets fit within the primary memory of a large cloud multicore.

A multicore-centric programming environment provides many advantages relative to distributed computing and GPUs. A large distributed system may have tremendous computational resources at its disposal, but its raw power comes with added complexity and overhead. Data must be moved over the network, and the system must support a degree of fault tolerance. These overheads tend to be small for problems that are embarrassingly parallel, but can come to dominate the execution time of computations that need to operate on shared data. Additionally, the multicore programming environment provides a degree of programming flexibility that can enable the rapid development of novel algorithms and software systems for emerging applications in industry and the sciences.

Despite multicores being readily available and providing a powerful general-purpose programming environment, the notorious difficulty of parallel programming has blunted their potential. For the scientific computing community, developing quality software systems that can take full advantage of the capabilities of large multicore hardware is a costly endeavour: requiring more time, more expert programmers, and more complex software systems. The complexity of writing performant multicore programs has generated considerable stress in industry and academia:

"When we start talking about parallelism and the use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced." — *John Hennessy, Turing Award Winner, in "A conversation with John Hennessy and David Patterson"* [150]

The complexity of parallel programming has made it more difficult for even expert programmers to develop high-quality multicore software systems. For average programmers, developing parallel programs that are debuggable, correct, and performant is a daunting challenge. This thesis is concerned with the development of programming technologies that reduce the complexity of parallel programming to make it easier for average programmers to exploit the capabilities of multicore hardware.

Programming technology for quality multicore software

In this thesis, I seek to simplify the engineering of multicore software systems by making it easier for programmers to write ***quality code*** that has simple understandable structure and performs well in practice. In pursuit of this goal, I have sought to develop technologies that simplify the theory and practice of parallel programming. These programming technologies span a broad range that includes parallel algorithms, data structures, optimization techniques, profiling tools, and system design principles. I contend that these technologies, by simplifying the design and performance engineering of parallel code, can empower average programmers to productively develop quality multicore software. Along these ends, this thesis provides evidence that appropriate programming technologies can simplify the design and performance engineering of parallel code.

Specifically, the artifacts in this thesis are in support of the following statement.

> **Thesis statement:** Parallel software systems designed using appropriate programming technologies can have a simple understandable structure and achieve good performance in practice.

In support of my thesis statement, I present seven principal intellectual artifacts that advance the state-of-the-art in the domains of parallel algorithms, multicore-centric systems for scientific computing, and programming tools for understanding and optimizing performance in parallel systems:

- **Chromatic** (Chapter 2): A parallel algorithm for scheduling data-graph computations deterministically. From "Executing dynamic data-graph computations deterministically using chromatic scheduling" published in SPAA 2014 [184] and in TOPC 2016 [183] with coauthors: William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson.

- **Color** (Chapter 3): Parallel algorithms and ordering heuristics for graph coloring that have the simple semantics of serial code. From "Ordering heuristics for parallel graph coloring" published in SPAA 2014 [146] with coauthors: William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson.

- **PARAD** (Chapter 4): An efficient and parallelism-preserving algorithm for performing automatic differentiation in parallel programs. From "PARAD: A work-efficient parallel algorithm for reverse-mode automatic differentiation" in-submission with coauthors: Tao B. Schardl, Brian Xie, Jie Chen, Aldo Pareja, Georgios Kollias, and Charles E. Leiserson.

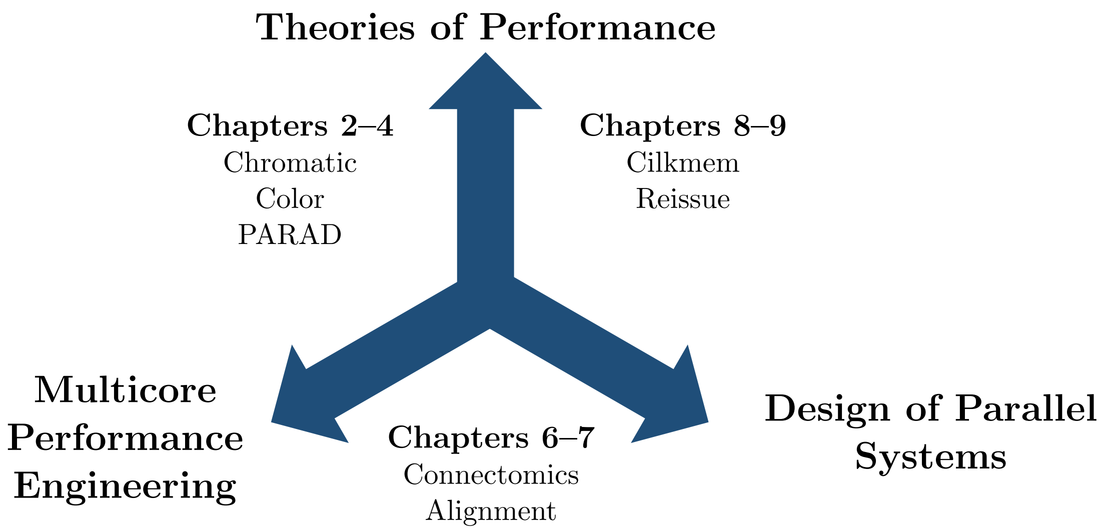


Figure 1-1: Graphical illustration of this thesis's seven principal artifacts: Chromatic, Color, PARAD, Connectomics, Alignment, Cilkmem, and Reissue.

- **Connectomics** (Chapter 5): An end-to-end image segmentation and skeletonization pipeline for connectomics using a single large multicore. From "A multicore path to connectomics-on-demand" published in PPoPP 2017 [247] with coauthors: Alexander Matveev, Yaron Meirovitch, Hayk Saribekyan, Wiktor Jakubiuk, Gergely Odor, David Budden, Aleksandar Zlateski, and Nir Shavit.

- **Alignment** (Chapter 6): An image-alignment pipeline for connectomics that uses memory-efficient algorithms, and techniques for judiciously exploiting performance–accuracy tradeoffs. From "High-throughput image alignment for connectomics using frugal snap judgments" published in HPEC 2020 [188] with coauthors: Brian Wheatman and Sarah Wooders.

- **Cilkmem** (Chapter 7): Efficient algorithms and tools for measuring the worst-case memory high-water mark of parallel programs. From "Cilkmem: Algorithms for analyzing the memory high-water mark of fork-join parallel programs" published in APOCS 2020 [186] with coauthors: William Kuszmaul, Tao B. Schardl, and Daniele Vettorel.

- **Reissue** (Chapter 8): Reissue policies for reducing tail latency in distributed services that are easy to analyze and effective in practice. From "Optimal reissue policies for reducing tail latency" published in SPAA 2017 [185] with coauthors: Yuxiong He and Sameh Elnikety.

Organization

The seven principal artifacts of my thesis fall into roughly three categories based on each artifact's approach to simplifying the development of quality code. Figure 1-1 illustrates the organization of the thesis's principal artifacts. The contributions to parallel algorithms employ techniques relating to theoretical models of performance and multicore performance engineering to obtain easy-to-understand code that performs well in practice. The contributions to scientific computing systems employ techniques relating to multicore performance engineering and parallel system design to obtain simple and high-performance software systems. The contributions to programming tools use theories of performance in conjunction with parallel system design to understand and optimize anomalous behavior relating to memory usage and latency in parallel systems. I summarize, in the following paragraphs, these three categories of artifacts and explain how they support the development of quality code.

Parallel algorithms. Chapters 2–4 present advancements in parallel algorithms for data-graph computations, graph coloring, and automatic differentiation. These artifacts support the development of quality code by showing how parallel algorithms can be designed to match the simple semantics of a sequential program while being efficient and scalable both in theory and in practice. Chapter 2 (the Chromatic artifact) shows how carefully designed parallel scheduling algorithms for data-graph computations can preclude the need for nondeterministic synchronization using locks. Chapter 3 (the Color artifact) introduces ordering heuristics for parallel graph coloring that are deterministic, provably scalable, and produce graph colorings of similar quality to known ordering heuristics used in serial code. Chapter 4 (the PARAD artifact) shows how serial algorithms for performing automatic differentiation can be generalized to parallel code in a work-efficient and parallelism-preserving manner. In Section 1.1, I provide background on shared-memory multicore programming, explain why determinism and serial semantics are critical in quality multicore code, and discuss this thesis's contributions to parallel algorithms in greater depth.

Multicore-centric scientific computing. Chapters 5–6 present case studies on the design and performance engineering of software systems that solve computational problems in the field of connectomics. These artifacts support the development of quality code by illustrating how simply designed parallel systems can achieve state-of-the-art performance through the use of quality parallel algorithms and simple performance engineering techniques. Chapter 5 (the Connectomics artifact) discusses the design and implementation of an image reconstruction pipeline. In this system, I show how careful parallelization and performance engineering can allow a single shared-memory multicore to outperform more complex systems that employed distributed computing and GPUs. Chapter 6 (the Alignment artifact) discusses a multicore-centric image-alignment pipeline for connectomics. In this system, I show how carefully designed multicore software systems can scale vertically to larger machines, horizontally over many multicores in a cluster, and scale to the largest conceivable data sets on the horizon in connectomics using machines with less than 1 TB of memory. In Section 1.2 I provide background on the field of connectomics, and summarize this thesis's contributions in the area of multicore-centric scientific computing for image reconstruction and image alignment in connectomics.

Tools for understanding parallel systems. Chapters 7–8 present technologies for understanding and mitigating anomalous behavior in parallel systems. These artifacts support the development of quality software by providing principled methods for understanding the interaction between a system's structure and its anomalous, or worst-case, behavior. Chapter 7 (the Cilkmem artifact) presents new algorithms for computing the exact and approximate worst-case memory usage of multicore programs. Cilkmem enables a programmer to understand the worst-case memory usage of their multicore code, which may only be observed in practice rarely, by running the program once under Cilkmem's instrumentation. Chapter 8 (the Reissue artifact) discusses strategies for mitigating the impact of "stragglers" in distributed request-response workflows. In such distributed systems, a key metric to optimize is the 99th percentile tail-latency of a request, which is often accomplished by replicating the responding service and judiciously sending duplicate requests to different replicas. The Reissue artifact describes and theoretically compares families of policies for sending these duplicate requests and analyzes their ability to effectively reduce tail latency. Section 1.3 discusses this thesis's contributions to programming tools for analyzing and optimizing worst-case memory usage and tail latency in parallel systems in greater depth.

In addition to the seven principal artifacts, Chapter 9 (the Retroactive artifact) presents a fully retroactive priority queue data structure with polylogarithmic overheads. The contributions in the Retroactive artifact are summarized in Section 1.4. Although this artifact does not directly relate to quality code, the data structure techniques employed to solve this open problem in retroactive data structures were inspired by techniques used to manage parallel data structures.

Section 1.5 provides an overview of the scope and impact of the artifacts in this thesis.

1.1 Shared-memory multicore programming

This section discusses techniques for reducing the complexity of writing correct and efficient shared-memory multicore code. I discuss the observation, made by many other researchers in the field, that parallel programming is substantially less complex when it is deterministic and has the semantics of serial code. I summarize this thesis's contributions in Chapters 2–4 to parallel algorithms and data structures for graph coloring, data-graph computations, and reverse-mode automatic differentiation.

Parallelism with serial semantics

Developing correct and performant parallel programs is substantially more complex than writing sequential code. The patterns and invariants of sequential code that programmers rely upon when writing software are often lost when parallelism is introduced. One can not easily rationalize the correctness of most parallel programs by walking through its steps on a napkin. Nor can one know whether repeated runs of a parallel program on the same input will yield the same, or even a correct, result. A programmer cannot even do something as simple as push elements to a dynamic array in parallel without tackling the complexities of concurrent programming.

> "Although threads seem to be a small step from sequential computations, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computations: understandability, predictability, and determinism." *Edward A. Lee in "The Problem with Threads"* [211].

The design of abstractions that allow programmers to productively write correct, performant, and maintainable multicore code is a key challenge in the parallel computing community [12]. It has been convincingly argued that these abstractions are most useful when they encapsulate nondeterminism due to concurrency and maintain the semantics of serial code [250, 43, 211, 33]:

"The aggressive goal of the parallel revolution is to make it as easy to write programs that are efficient, portable, and correct [...] as it has been to write programs for sequential computers." *Krste Asanovic et al. in "A view of the parallel computing landscape"* [12].

Many researchers over multiple decades have advocated that the difficulty of parallel programming can be greatly reduced by using some form of "deterministic parallelism" [281, 143, 125, 326, 32, 106, 105, 92, 93, 160, 23, 24, 274, 353, 43]. With a deterministic parallel program, the programmer observes no logical ***concurrency***, that is, no nondeterminacy in the behavior of the program due to the relative and nondeterministic timing of communicating processes such as occurs when one process arrives at a lock before another. The semantics of a deterministic parallel program are therefore serial, and reasoning about such a program's correctness, at least in theory, is no harder than reasoning about the correctness of a serial program. Testing, debugging, and formal verification is simplified, because there is no need to consider all possible relative timings (interleavings) of operations on shared mutable data.

Despite the apparent advantages of deterministic parallelism, however, most parallel programs deployed in practice exhibit non-deterministic behavior due to concurrency. Indeed, most parallel programs today are still written using Pthreads [168], which forces programmers to use concurrency mechanisms such as mutex locks and condition variables. As a consequence, only experts can program these parallel applications, especially those for shared-memory platforms, and these bug-prone codes can only be understood by experts. For example, all the codes in the PARSEC [27], Galois [283], and STAMP [60] benchmark suites use concurrency mechanisms.

Nevertheless, important steps toward determinism have been made. Perhaps the most significant practical progress has occurred in the realm of ***fork-join parallelism***. Fork-join parallelism is usually implemented using ***work-stealing*** [57, 103, 108, 114, 143, 189, 199, 202, 270, 337, 41, 39, 116], where worker threads in the runtime system coordinate to load-balance the computation, as in the various Cilk dialects [38, 116, 82, 219, 212, 172], Fortress [3], Habanero [16], Habanero-Java [61], Hood [42], HotSLAW [258], Java Fork/Join Framework [208], OpenMP [275, 13], Task Parallel Library [218], Threading Building Blocks (TBB) [291] and X10 [68]. In this model, subroutines can be spawned in parallel, generating a series-parallel execution dag in which the synchronization of subtasks is managed "under the covers" by the runtime system. Constructs such as `parallel_for` can be implemented as syntactic sugar on top of the fork-join model. As long as the parallel program contains no ***determinacy races*** [106] (also called ***general races*** [266]), the program is deterministic. Moreover, efficient tools exist that can guarantee to detect determinacy races or validate their absence [106, 107].

Chromatic (Chapter 2)

The Chromatic artifact discusses the deterministic schedulers for data-graph computations Prism and Prism-R. Data-graph computations have been popularized in such programming systems as Galois [267, 268], Pregel [242], GraphLab [234, 233], PowerGraph [134], Ligra [318, 321], and GraphChi [203]. Prism employs a technique called chromatic scheduling [26, 1, 233] to resolve data races in a data-graph computation deterministically. Prism computes a vertex coloring of the graph that it uses to coordinate updates performed in a round, precluding the need for mutual exclusion locks or other nondeterministic synchronization. Surprisingly, Prism outperforms existing lock-based schedulers, showing that the "price of determinism" in the case of scheduling data-graph computations can be negative.

A key data structure challenge solved in the Chromatic artifact is the efficient and deterministic management of dynamic collections in parallel code. During a data-graph computation, Prism maintains dynamic sets of vertices that are partitioned by color called ***color sets***. The Chromatic artifact introduces two data structures for representing these color sets: the "multibag" and "multivector." The ***multibag*** provides an efficient way to manage color sets in parallel, but does not guarantee that the ordering of vertices within a color set will be deterministic. The ***multivector*** is similar to the multibag but additionally ensures that the vertices in each color set are ordered deterministically. In fact, the multivector maintains the same semantics as a set of dynamic arrays that support the insertion of vertices via an append operation — vertices are ordered based on the order they would have been inserted in a serial execution.

The Prism-R algorithm extends Prism to handle data-graph computations whose update functions perform global associative reductions. Prism-R uses a multivector to represent its color sets to ensure that vertices of the same color are deterministically ordered. The multivector enables Prism-R to guarantee that global associative reductions performed over vertices in the graph have a deterministic result. Prism-R and the multivector provide these strong serial semantics while matching the theoretical bounds on work, span, and parallelism achieved by Prism. In practice, Prism-R has only a 7% geometric mean overhead relative to Prism on seven application benchmarks.

Color (Chapter 3)

The Color artifact discusses the design of vertex-ordering heuristics for parallel graph coloring algorithms. Ordering heuristics are vital to achieving quality vertex-colorings when using a greedy algorithm. In fact, the first ordering heuristic called largest-degree-first (LF) was described by Welsh and Powell [341] alongside their introduction of the first greedy graph coloring algorithm in 1967. Numerous other orderings have been proposed since then such as smallest-degree-last (SL) [246, 4], incidence-degree (ID) [75], and saturation-degree (SD) [48] heuristics.

It may appear that an ordering heuristic for parallel graph-coloring must necessarily be aware of concurrency, but it turns out that ordering heuristics used in parallel graph coloring need not be concerned directly with the details of the parallel execution. Rather, the intrinsic parallelism of a particular ordering heuristic (as used in a serial code) can be analyzed theoretically and then directly applied using a deterministic parallel graph coloring code that matches the result of the serial code.

In Chapter 3, we introduce the largest-log-degree-first (LLF) and smallest-log-degree-last (SLL) ordering heuristics for parallel greedy graph-coloring algorithms, which are inspired by the largest-degree-first (LF) and smallest-degree-last (SL) serial heuristics, respectively. We show that although LF and SL, in practice, generate colorings with relatively small numbers of colors, they are vulnerable to adversarial inputs for which any parallelization yields a poor parallel speedup. In contrast, LLF and SLL allow for provably good speedups on arbitrary inputs while, in practice, producing colorings of competitive quality to their serial analogs.

PARAD (Chapter 4)

The PARAD artifact presents the first work-efficient parallelism-preserving algorithm for performing reverse-mode automatic differentiation. Automatic differentiation (AD) is a technique for computing the derivative of function $F : \mathbb{R}^n \to \mathbb{R}^m$ defined by a computer program. Modern applications of AD, such as machine learning, typically use AD to facilitate gradient-based optimization of an objective function for which $m \ll n$ (often $m = 1$). As a result, these applications typically use reverse (or adjoint) mode AD to compute the gradient of F efficiently, in time $\Theta(m \cdot T_1(F))$, where $T_1(F)$ is the work (serial running time) of F. Although the serial running time of reverse-mode AD has a well known relationship to the total work of F, general-purpose reverse-mode AD has proven challenging to parallelize in a work-efficient and scalable fashion, as simple approaches tend to result in poor performance or scalability.

PARAD is a work-efficient parallel algorithm for reverse-mode AD of recursive fork-join programs. We analyze the performance of PARAD using work-span analysis. Given a program F with work $T_1(F)$ and span (critical-path length) $T_\infty(F)$, PARAD performs reverse-mode AD of F in $O(m \cdot T_1(F))$ work and $O(\log m + \log(T_1(F))T_\infty(F))$ span. To the best of our knowledge, PARAD is the first parallel algorithm for performing reverse-mode AD that is both provably work-efficient and has span within a polylogarithmic factor of the original program F.

We implemented PARAD as an extension of Adept, a C++ library for performing reverse-mode AD for serial programs, which is known for its efficiency. Our implementation supports the use of Cilk fork-join parallelism and requires no programmer annotations of parallel control flow. Instead, it uses compiler instrumentation to dynamically trace a program's series-parallel structure, which is used to automatically parallelize the gradient computation via reverse-mode AD. On eight machine-learning benchmarks, our implementation of PARAD achieves 1.5× geometric-mean multiplicative work overhead relative to the serial Adept tool, and 8.9× geometric-mean self-relative speedup on 18 cores.

1.2 Multicore-centric systems for scientific computing

This section summarizes this thesis's contributions in Chapters 5–6 that simplify the engineering of high-performance systems for scientific computing. Specifically, this thesis presents work on two software systems that solve computational problems arising in the field of connectomics using a multicore-centric approach. Surprisingly, we find that a mixture of general and domain-specific performance optimization techniques can be used to obtain performance on a single multicore that rivals that obtained when using large clusters of CPUs and GPUs.

Background on connectomics

The field of connectomics uses cutting edge machine learning and image processing to extract brain connectivity graphs from electron microscopy images. Advances in electron microscopy have enabled the acquisition of image data sets that capture both the small and large scale features present in neural tissue. The resultant data sets are quite large with a relatively small $1mm^3$ volume producing petabytes of data when imaged at $3 \times 3 \times 30$nm resolution. The scale of the acquired data necessitate the development of image processing systems that are both scalable and efficient.

It has long been assumed that the processing of connectomics data will require mass storage, farms of CPU/GPUs, and months (if not years) of processing time. However, we present in Chapter 5 and Chapter 6 two multicore-centric systems for connectomics that can match the terabyte-per-hour pace of modern electron microscopes while using less than 100 cores.

Connectomics pipeline (Chapter 5)

The Connectomics artifact presents a case study of the design of a large-scale image segmentation pipeline for connectomics that is designed to run on a single multicore server. By eschewing the complexities involved in the design of a distributed system, this multicore-centric pipeline was able to avoid overheads related to communication, fault-tolerance, and data serialization. Perhaps more importantly, the decision to target a single multicore allowed us to focus entirely on the design and optimization of shared-memory multicore algorithms. These performance optimizations turned out to be extremely impactful, and allowed our single 72-core multicore server to process a terabyte of data in 4 hours which outperformed the previous fastest system that used a cluster of 512 cores to process a terabyte of data in 140 hours.

The multicore-centric segmentation pipeline presented in Chapter 5 provides compelling evidence that the benefits of quality algorithms, parallelization, and careful engineering can sometimes be so vast that it is possible to solve "cluster-scale" problems on a single commodity multicore machine. The results presented in the Connectomics artifact push back against the current design trends in large-scale machine-learning that emphasize the ability to scale across large clusters of CPUs and GPUs.

Alignment pipeline (Chapter 6)

The Alignment artifact presents a high-throughput image alignment pipeline for connectomics that employs the multicore algorithms Quilter and Stacker to perform 2D and 3D alignment, respectively. As part of the optimization of this pipeline, this chapter introduces a technique for data-driven performance optimization called "frugal snap judgments" that is used to obtain more advantageous performance–accuracy trade-offs in Quilter.

We introduce the algorithms Quilter and Stacker that are designed to perform 2D and 3D alignment, respectively, on petabyte-scale data sets from connectomics. Quilter and Stacker are efficient, scalable, and simple to deploy on hardware ranging from a researcher's laptop to a large-scale computing cluster. On a single 18-core cloud machine, each algorithm achieves throughputs of more than 1 TB/hr and, when combined, produce an end-to-end alignment pipeline that processes data at a rate of 0.82 TB/hr — an over 10x improvement over previous systems. This efficiency comes from both traditional optimizations and from the use of "Frugal Snap Judgments" to judiciously exploit performance–accuracy trade-offs. A high-throughput image-alignment pipeline was implemented and evaluated using the Quilter and Stacker algorithms. The performance was evaluated on a range of platforms including a common 18-core machine (Intel E5), a large 112-core machine (Intel Xeon Platinum), and a supercomputing cluster with 1600 cores. The pipeline achieves a throughput of 0.6–0.8 TB/hr on the 18-core machine, 1.4–1.5 TB/hr on the large 112-core machine, and 21.4 TB/hr on the supercomputing cluster with 1600 cores.

The results in Chapter 6 illustrate how carefully designed multicore software components can yield software systems that perform well on individual machines and also support simple horizontal scaling over multiple machines in a computing cluster. Furthermore, it illustrates how higher-level performance optimization techniques, such as frugal snap judgments, can be employed to dramatically improve performance without substantially increasing system complexity. The data-driven performance optimizations employed using frugal snap judgments are conceptually simple, yet yield significant performance improvements.

1.3 Beyond runtime: Tools for bounding memory usage and tail latency

This section describes this thesis's contributions in Chapters 7–8 that develop tools and techniques for analyzing and optimizing anomalous behavior related to memory and tail latency in parallel systems. Performance anomalies that are rarely seen in a small software system can become harmfully common when deploying the system at scale. In the Reissue and Cilkmem artifacts we focus on two performance phenomena that require consideration when developing large-scale software systems: worst-case memory usage in multicore codes and response-time tail-latency in distributed request-response workflows.

Cilkmem (Chapter 7)

The Cilkmem artifact presents a tool and algorithms for measuring a parallel program's worst-case memory high-water mark. Understanding the worst-case memory high-water mark of a parallel program is of interest to software engineers designing software that has high memory requirements or executes in memory-constrained environments. Software engineers designing such programs must be cognizant of how their program's memory requirements scale in a many-processor execution. Although tools exist for measuring memory usage during one particular execution of a parallel program, such tools cannot bound the worst-case memory usage over all possible parallel executions.

The Cilkmem tool analyzes the execution of a deterministic Cilk program to determine its p-processor memory high-water mark (MHWM), which is the worst-case memory usage of the program over all possible p-processor executions. Cilkmem employs two new algorithms for computing the p-processor MHWM. The first algorithm calculates the exact p-processor MHWM in $O(T_1 \cdot p)$ time, where T_1 is the total work of the program. The second algorithm solves, in $O(T_1)$ time, the approximate threshold problem, which asks, for a given memory threshold M, whether the p-processor MHWM exceeds $M/2$ or whether it is guaranteed to be less than M. Both algorithms are memory efficient, requiring $O(p \cdot D)$ and $O(D)$ space, respectively, where D is the maximum call-stack depth of the program's execution on a single thread.

Our empirical studies show that Cilkmem generally exhibits low overheads. Across ten application benchmarks from the Cilkbench suite, the exact algorithm incurs a geometric-mean multiplicative overhead of 1.54 for $p = 128$, whereas the approximation-threshold algorithm incurs an overhead of 1.36 independent of p. In addition, we use Cilkmem to reveal and diagnose a previously unknown issue in a large image-alignment program contributing to unexpectedly high memory usage under parallel executions.

Reissue (Chapter 8)

The Reissue artifact presents a principled strategy for reducing tail latency in distributed request-response workflows by judiciously sending duplicate copies of requests. Sending duplicate requests to replicated services is a simple and commonly used strategy in distributed systems for reducing latency. Determining the optimal strategy for sending duplicate requests, however, is challenged by the difficulty of obtaining closed-form solutions to complex problems in queueing theory. Data-driven methods are often employed to tune a reissue policy's parameters, in practice, to find good reissuing strategies. It is not known, however, the degree to which the parametrization of a reissue policy impacts its performance. In Chapter 8, we make progress towards understanding the relative power of different parameterized families of reissue policies within a simplified analytical model, and corroborate these conclusions with a combination of simulation and real-world experiments.

Interactive distributed services send redundant requests to multiple different replicas to meet stringent tail latency requirements. These additional (reissue) requests mitigate the impact of nondeterministic delays within the system and thus increase the probability of receiving an on-time response. There are two existing approaches to using reissue requests to reduce tail latency. (1) Reissue requests immediately to one or more replicas, which multiplies the load and runs the risk of overloading the system. (2) Reissue requests if not completed after a fixed delay. The delay helps to bound the number of extra reissue requests, but it also reduces the chance for those requests to respond before a tail latency target.

We introduce a new family of reissue policies, **Single-Time / Random** (SingleR), that reissues requests after a delay d with probability q. SingleR employs randomness to bound the reissue rate, while allowing requests to be reissued early enough to have sufficient time to respond, exploiting the benefits of both immediate and delayed reissue of prior work. We formally prove, within a simplified analytical model, that SingleR is optimal even when compared to more complex policies that reissue multiple times. In a set of simulations and experiments on real-world systems, we show that SingleR policies are effective in practice. For example, SingleR reduces the 99th-percentile latency of Redis by 30–70% by reissuing only 2% of requests, and the 99th-percentile latency of Lucene search by 15–25% by reissuing only 1% of requests.

1.4 Other contributions

In addition to the principal artifacts in this thesis, Chapter 9 (the Retroactive artifact) presents a result that resolves an open problem in retroactive data structures using techniques that were inspired by, but do not directly relate to, the design of deterministic parallel data structures. Specifically, the "hierarchical checkpointing" technique employed in the Retroactive paper was inspired by work relating to the design of reducer hyperobjects [115].

Retroactive (Chapter 9)

The Retroactive artifact describes a fully retroactive priority queue data structure that has polylogarithmic overheads. Although this artifact is not directly concerned with parallel programming, the techniques used to solve this previously open problem were inspired by those used to devise efficient parallel data structures.

Since the introduction of retroactive data structures at SODA 2004, a major open question has been the difference between partial retroactivity (where updates can be made in the past) and full retroactivity (where queries can also be made in the past). In particular, for priority queues, partial retroactivity is possible in $O(\log m)$ time per operation on an

m-operation timeline, but the best previously known fully retroactive priority queue has cost $\Theta(\sqrt{m}\log m)$ time per operation.

In Chapter 9, I address this open problem by providing a general logarithmic-overhead transformation from partial to full retroactivity called hierarchical checkpointing, provided that the given data structure is time-fusible (multiple structures with disjoint timespans can be fused into a timeline supporting queries of the present). As an application, we construct a fully retroactive priority queue which can insert an element, delete the minimum element, and find the minimum element, at any point in time, in $O(\log^2 m)$ amortized time per update and $O(\log^2 m \log\log m)$ time per query, using $O(m \log m)$ space.

1.5 Overview

The overall goal of this thesis is to develop programming technologies that facilitate the development of quality multicore code — code that has simple understandable structure and performs well in practice. These programming technologies are well suited to a broad set of programmers since they are simple to use, have theoretical guarantees, and perform well in practice. This section explains why I believe it is important to (a) improve the programmability of shared-memory multicores and, (b) emphasize technologies that enable programmers to write quality code. The section ends with a concise outline of the remaining chapters of this thesis which are adapted from my published articles.

Scope

The primary scope of this thesis lies in the realm of shared-memory multicore programming. Six of the seven principal artifacts in this thesis directly concern algorithms, data structures, and software systems for shared-memory multicore machines. The Reissue artifact is not directly concerned with multicore programming, but relates to problems arising in large-scale distributed systems.

The parallel algorithms described in this thesis are all designed using fork-join parallelism and analyzed in the dag model of multithreading [40, 41] using work-span analysis [77, Ch. 27]. The requisite background information relating to the parallel linguistics and runtime are provided within each chapter of the thesis.

Distributed computing, GPUs, and specialized hardware

Parallel computing can be performed using platforms other than shared-memory multicores including distributed systems, GPUs, FPGAs, and other specialized hardware. Distributed computing is especially helpful when tackling problems that do not fit in-memory on a single multicore, but they encounter overheads relating to fault-tolerance and communication of intermediate results. GPUs, FPGAs, and other hardware accelerators (e.g. TPUs) tend to be more memory-constrained than multicores, are substantially more complex to program, and are generally suited to a more narrow range of problems.

Within this thesis, the benefits of multicore-centric computing relative to GPUs and distributed computing are illustrated in the Connectomics and Alignment artifacts (Chapters 6–7). Chapter 5 (the Connectomics artifact) provides an extreme case in which a single multicore with superiorly optimized software outperformed much larger systems that used distributed computing and GPUs. Chapters 6–7 provide multiple exhibits that illustrate

Method	Type	8-channel MaxoutNet Throughput (MB/s)	32-channel MaxoutNet Throughput (MB/s)
cpuXNN	CPU (72-core)	111.1	16.67
gpuZNN	GPU (Titan X)	67.61	25.28
gpuNeon	GPU (Titan X)	37.06	(exceeds memory)

Figure 1-2: A comparison of CNN throughput for the best-performing CPU and GPU-based implementations using the MaxoutNet architecture.

2D Alignment Method	Runtime	Throughput
FijiBento	362 minutes	0.091 TB/hr
Quilter Full Resolution	180 minutes	0.18 TB/hr
Quilter FSJ(20,100)	32 minutes	1.03 TB/hr

Figure 1-3: Performance comparison of FijiBento and Quilter for 2D Alignment on the *Common Multicore* platform, an 18-core AWS C4 instance.

the power of properly optimized multicore software, but let us examine just two illustrative examples.

The Connectomics artifact (Chapter 5) includes a direct performance comparison of CPU and GPU implementations of a CNN architecture for membrane detection. The implementations that are compared include a 72-core multicore implementation, a custom GPU implementation, and a GPU implementation that uses existing software libraries for executing CNNs. Figure 1-2 compares GPU and multicore implementations of the CNN *MaxoutNet* architecture that is used in the Connectomics artifact. The *gpuNeon* implementation used existing GPU libraries for computing CNNs and underperformed our 72-core multicore implementation by 3x on the 8-channel *MaxoutNet* architecture. Worse still, *gpuNeon* failed to execute the 32-channel network due to memory constraints. The *gpuZNN* provides an optimized GPU implementation that outperforms the multicore implementation by 1.5x on the 32-channel architecture, but is 1.6x worse on the 8-channel architecture which is what the software pipeline in the Connectomics artifact actually uses. This example demonstrates that shared-memory multicore code is not, necessarily, substantially slower than specialized hardware. In fact, properly optimized multicore code can outperform GPUs in certain cases, such as in the 8-channel architecture, where the low amount of compute performed per-byte causes data transfers to the GPU to bottleneck performance.

The Alignment (Chapter 6) artifact includes a comparison between a distributed and shared-memory approach for solving the 2D alignment problem in connectomics. The overheads of distributed computing are illustrated in Figure 1-3 from the Alignment artifact. The FijiBento software system employs distributed computing by partitioning the task of 2D aligning an image into many fine-grained tasks, whereas Quilter parallelizes the computation in-memory. Even when FijiBento runs on a single 18-core machine, precluding the need to communicate intermediate results over the network, it performs 2x slower than Quilter due to overheads related to the serialization of intermediate results from its fine-grained tasks. Furthermore, the flexibility of the shared-memory programming model allows us to implement optimizations relating to performance–accuracy trade-offs in Quilter FSJ(20,100) that improve performance by an additional 5x. Such performance–accuracy trade-offs are not as valuable in the distributed setting since it is more difficult to implement fine-grained dynamic control flow, and the overheads of data serialization reduces the relative value of optimizing the application-specific, as opposed to compute-framework, logic.

For the reasons outlined here, I believe that in the present moment shared-memory multicores strike the best balance between general programmability and performance. This belief motivates the focus of this thesis on the development of programming technologies which reduce the complexity of parallel programming on shared-memory multicores.

The value of quality multicore code

Why is it important for code and programming technology to have a simple understandable structure? I believe that the reason goes beyond simple aesthetics. I contend that the simple understandable structure of quality code has tangible value. Quality code is able to more readily incorporate refined programming patterns from the past, and be understood across disciplines.

Techniques for developing parallel code with simple understandable structure and good performance in practice broadens the set of algorithms that can be efficiently brought into the modern era. A few examples can be observed in the artifacts of this thesis. The Chromatic artifact was used to parallelize a provably good three-dimensional thinning algorithm developed in 1995 [25]. This parallel implementation was directly used in the Connectomics artifact to perform skeletonization of neural volumes while provably preserving their topology. The Color artifact designed ordering heuristics for parallel graph coloring that were directly inspired by serial heuristics developed between 1967 and 1995 [341, 75, 246, 4, 48]. The Color artifact evaluates the theoretical parallelism in these heuristics in a principled fashion and shows how many of these heuristics can be coarsened to be theoretically scalable and achieve good coloring qualities in practice. The PARAD artifact provides a work-efficient algorithm for parallel automatic differentiation which extends the simple serial automatic differentiation algorithms that have been known since the 1960s [342, 230, 324].

Quality code with simple understandable structure can be more readily understood and adapted by practitioners in other scientific or industrial fields. Performance–accuracy trade-offs are exploited in the Alignment artifact to achieve a 5x improvement to performance using relatively simple machine-learning techniques that learn criteria to switch between "fast" and "slow" code paths. The Reissue artifact shows how to analyze reissue policies for distributed request-response workflows in a simplified analytical model. The Cilkmem artifact shows how to analyze the worst-case memory usage of a fork-join parallel program. This enables practitioners using shared computing clusters to correctly reason about the amount of memory they must reserve for their multicore task.

Summary

The goal of this thesis is to make it easier to develop parallel software systems that have a simple understandable structure and achieve good performance in practice. My approach in this thesis has mostly focused on the development of programming technologies that improve the simplicity and performance of parallel computing in the shared-memory multicore setting. Although there are other platforms in which parallelism can be expressed, such as distributed computing and GPUs, I believe there is compelling evidence that the flexible programming environment provided by general-purpose multicores makes them well-suited to many computational problems in science and industry. I have also discussed ways in which the simple understandable structure of quality code has tangible value by enabling it to better leverage and incorporate algorithmic insights developed in sequential programming models.

Chapter 2

Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling

This chapter presents the PRISM and PRISM-R algorithms for executing dynamic data-graph computations deterministically using chromatic scheduling. The Multibag and Multivector data structures are presented as key data structures that allow PRISM and PRISM-R to maintain active sets of vertices in a work-efficient manner and preserve serial semnatics. This work was conducted in collaboration with William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson.

Abstract

A ***data-graph computation*** — popularized by such programming systems as Galois, Pregel, GraphLab, PowerGraph, and GraphChi — is an algorithm that performs local updates on the vertices of a graph. During each round of a data-graph computation, an ***update function*** atomically modifies the data associated with a vertex as a function of the vertex's prior data and that of adjacent vertices. A ***dynamic*** data-graph computation updates only an active subset of the vertices during a round, and those updates determine the set of active vertices for the next round.

This chapter introduces PRISM, a chromatic-scheduling algorithm for executing dynamic data-graph computations. PRISM uses a vertex-coloring of the graph to coordinate updates performed in a round, precluding the need for mutual-exclusion locks or other nondeterministic data synchronization. A ***multibag*** data structure is used by PRISM to maintain a dynamic set of active vertices as an unordered set partitioned by color. We analyze PRISM using work-span analysis. Let $G = (V, E)$ be a degree-Δ graph colored with χ colors, and suppose that $Q \subseteq V$ is the set of active vertices in a round. Define $\mathit{size}(Q) = |Q| + \sum_{v \in Q} \deg(v)$, which is proportional to the space required to store the vertices of Q using a sparse-graph layout. We show that a P-processor execution of PRISM performs updates in Q using $O(\chi(\lg(Q/\chi) + \lg \Delta) + \lg P)$ span and $\Theta(\mathit{size}(Q) + P)$ work.

These theoretical guarantees are matched by good empirical performance. In order to isolate the effect of the scheduling algorithm on performance, we modified GraphLab to incorporate PRISM and studied seven application benchmarks on a 12-core multicore machine. PRISM executes the benchmarks 1.2–2.1 times faster than GraphLab's nondeter-

ministic lock-based scheduler while providing deterministic behavior.

This chapter also presents PRISM-R, a variation of PRISM that executes dynamic data-graph computations deterministically even when updates modify global variables with associative operations. PRISM-R satisfies the same theoretical bounds as PRISM, but its implementation is more involved, incorporating a ***multivector*** data structure to maintain a deterministically ordered set of vertices partitioned by color. Despite its additional complexity, PRISM-R is only marginally slower than PRISM. On the seven application benchmarks studied, PRISM-R incurs a 7% geometric mean overhead relative to PRISM.

2.1 Introduction

Many systems from physics, artificial intelligence, and scientific computing can be represented naturally as a ***data graph*** — a graph with data associated with its vertices and edges. For example, some physical systems can be decomposed into a finite number of elements whose interactions induce a graph. Probabilistic graphical models in artificial intelligence can be used to represent the dependency structure of a set of random variables. Sparse matrices can be interpreted as graphs for scientific computing.

A data-graph computation is an algorithm that performs "local" updates on the vertices of a data graph, taking as input data associated with a vertex and its neighbors. Several software systems have been implemented to support parallel data-graph computations, including GraphLab [234, 233], Pregel [242], Galois [267, 268], PowerGraph [134], Ligra[1] [318, 321], and GraphChi [203]. These systems can support "complex" data-graph computations, in which data can be associated with edges as well as vertices and updating a vertex v can modify any data associated with v, v's incident edges, and the vertices adjacent to v. For ease in discussing chromatic scheduling, however, we shall principally restrict ourselves to "simple" data-graph computations (which correspond to "edge-consistent" computations in GraphLab), although most of our results straightforwardly extend to more complex models. Indeed, six out of the seven GraphLab applications described in [234, 233] are simple data-graph computations.

Updates to vertices proceed in ***rounds***, where each vertex can be updated at most once per round. In a ***static*** data-graph computation, the ***activation set*** Q_r of vertices updated in a round r — the set of ***active*** vertices — is determined a priori. Often, a static data-graph computation updates every vertex in each round. Static data-graph computations include Gibbs sampling [124, 123], iterative graph coloring [81], and n-body problems such as the fluidanimate PARSEC benchmark [27].

We shall be interested in ***dynamic*** data-graph computations, where the activation set changes round by round. Dynamic data-graph computations include the Google PageRank algorithm [51], loopy belief propagation [263, 282], coordinate descent [90], co-EM [269], alternating least-squares [153], singular-value decomposition [133], and matrix factorization [333].

We formalize the computational model as follows. Let $G = (V, E)$ be a data graph. Denote the ***neighbors***, or ***adjacent vertices***, of a vertex $v \in V$ by $\text{Adj}[v] = \{u \in V : (u, v) \in E\}$. The ***degree*** of v is thus $\deg(v) = |\text{Adj}[v]|$, and the ***degree*** of G is $\deg(G) = \max\{\deg(v) : v \in V\}$. A ***(simple) dynamic data-graph computation*** is a

[1]While Ligra does not technically execute data-graph computations, it is designed to implement similar algorithms by decoupling the scheduling and algorithm-specific code, as with the other data-graph computation frameworks.

triple $\langle G, f, Q_0 \rangle$, where

- $G = (V, E)$ is an undirected graph with data associated with each vertex $v \in V$;
- $f : V \to 2^V$ is an ***update function***; and
- $Q_0 \subseteq V$ is the initial ***activation set***.

The update $S \leftarrow f(v)$ implicitly computes as a side effect a new value for the data associated with v as a function of the old data associated with v and v's neighbors. The update returns a set $S \subseteq \mathrm{Adj}[v]$ of vertices that must be updated in the next round. For example, an update $f(v)$ might activate a neighbor u only if the value of v changes significantly. During a round r of a dynamic data-graph computation, each vertex $v \in Q_r$ is updated at most once, that is, Q_r is a set, not a multiset.

The advantage of dynamic over static data-graph computations is that they avoid performing many unnecessary updates. Studies in the literature [234, 233] show that dynamic execution can enhance the practical performance of many applications. We confirmed these findings by implementing static and dynamic versions of several data-graph computations. The results for a PageRank algorithm on a power-law graph of 1 million vertices and 10 million edges were typical. The static computation performed approximately 15 million updates, whereas the dynamic version performed less than half that number of updates.

A serial reference implementation

Before we address the issues involved in scheduling and executing dynamic data-graph computations in parallel, let us first hone our intuition with a serial implementation. Figure 2-1 gives the pseudocode for SERIAL-DDGC. This algorithm schedules the updates of a data-graph computation by maintaining a FIFO queue Q of activated vertices that have yet to be updated. Sentinel values enqueued in Q on lines 4 and 9 demarcate the rounds of the computation such that the set of vertices in Q after the rth sentinel has been enqueued is the activation set Q_r for round r.

Given a data-graph $G = (V, E)$, an update function f, and an initial activation set Q_0, SERIAL-DDGC executes the data-graph computation $\langle G, f, Q_0 \rangle$ as follows. Lines 1–2 initialize Q to contain all vertices in Q_0. The **while** loop on lines 5–14 then repeatedly dequeues the next scheduled vertex $v \in Q$ on line 5 and executes the update $f(v)$ on line 11. Executing $f(v)$ produces a set S of activated vertices, and lines 12–14 check each vertex in S for membership in Q, enqueuing all vertices in S that are not already in Q.

We can analyze the time SERIAL-DDGC takes to execute one round r of the data-graph computation $\langle G, f, Q_0 \rangle$. Define the ***size*** of an activation set Q_r as

$$\mathit{size}(Q_r) = |Q_r| + \sum_{v \in Q_r} \deg(v) \; .$$

The size of Q_r is asymptotically the space needed to store all the vertices in Q_r and their incident edges using a standard sparse-graph representation, such as compressed-sparse-rows (CSR) format [328]. For example, if $Q_0 = V$, we have $\mathit{size}(Q_0) = |V| + 2|E|$ by the handshaking lemma [77, p. 1172–3]. Let us make the reasonable assumption that the time to execute $f(v)$ serially is proportional to $\deg(v)$. If we implement the queue as a dynamic (resizable) table [77, Section 17.4], then line 14 executes in $\Theta(1)$ amortized time. Of course, a linked list would suffice to append operations in $\Theta(1)$ time, but would not allow for convenient subsequent parallel iteration over its elements. All other operations in the **for** loop on lines 12–14 take $\Theta(1)$ time, and thus all vertices activated by executing

```
SERIAL-DDGC($G, f, Q_0$)
 1  for $v \in Q_0$
 2      ENQUEUE($Q, v$)
 3  $r \leftarrow 0$
 4  ENQUEUE($Q$, NIL) // Sentinel NIL denotes the end of a round.
 5  while $Q \neq \{\text{NIL}\}$
 6      $v \leftarrow$ DEQUEUE($Q$)
 7      if $v ==$ NIL
 8          $r$ += 1
 9          ENQUEUE($Q$, NIL)
10      else
11          $S \leftarrow f(v)$
12          for $u \in S$
13              if $u \notin Q$
14                  ENQUEUE($Q, u$)
```

Figure 2-1: Pseudocode for a serial algorithm to execute a data-graph computation $\langle G, f, Q_0 \rangle$. SERIAL-DDGC takes as input a data graph G and an update function f. The computation maintains a FIFO queue Q of activated vertices that have yet to be updated and sentinel values NIL, each of which demarcates the end of a round. An update $S \leftarrow f(v)$ returns the set $S \subseteq \text{Adj}[v]$ of vertices activated by that update. Each vertex $u \in S$ is added to Q if it is not currently scheduled for a future update.

$f(v)$ are examined in $\Theta(\deg(v))$ time. The total time spent updating the vertices in Q_r is therefore $\Theta(Q_r + \sum_{v \in Q_r} \deg(v)) = \Theta(size(Q_r))$, which is ***linear*** time: time proportional to the storage requirements for the vertices in Q_r and their incident edges.

Parallelizing dynamic data-graph computations

The salient challenge in parallelizing data-graph computations is to deal effectively with races between updates, that is, logically parallel updates that read and write common data. A ***determinacy race*** [105] (also called a ***general race*** [266]) occurs when two logically parallel instructions access the same memory location and at least one of them writes to that location. Two updates in a data-graph computation ***conflict*** if executing them in parallel produces a determinacy race. A parallel scheduler must manage or avoid conflicting updates to execute a data-graph computation correctly and deterministically.

The standard approach to preventing races associates a mutual-exclusion lock with each vertex of the data graph to ensure that an update on a vertex v does not proceed until all locks on v and v's neighbors have been acquired. Although this locking strategy prevents races, it can incur substantial overhead from lock acquisition and contention, hurting application performance, especially when update functions are simple. Moreover, because runtime happenstance can determine the order in which two logically parallel updates acquire locks, the data-graph computation can act nondeterministically: different runs on the same inputs can produce different results. Without repeatability, parallel programming is arguably much harder [211, 43]. Nondeterminism confounds debugging.

A known alternative to using locks is ***chromatic scheduling*** [26, 1, 233], which schedules a data-graph computation based on a coloring of the data-graph computation's ***conflict graph*** — a graph with an edge between two vertices if updating them in parallel would produce a race. For a simple data-graph computation, the conflict graph is simply the data

Benchmark	$\lvert V \rvert$	$\lvert E \rvert$	χ	RRLocks	Cilk+Locks	Prism	Prism-R
PR/G	916,428	5,105,040	43	15.5	14.3	9.7	12.6
PR/L	4,847,570	68,475,400	333	227.6	200.4	109.3	127.3
ID/2000	4,000,000	15,992,000	4	48.6	43.8	32.1	32.8
ID/4000	16,000,000	63,984,000	4	200.0	179.6	123.1	124.3
FBP/C1	87,831	265,204	2	8.7	8.9	6.9	7.0
FBP/C3	482,920	160,019	2	16.4	17.8	13.3	13.4
ALS/N	187,722	20,597,300	6	134.3	123.6	105.2	105.7

Figure 2-2: Comparison of dynamic data-graph schedulers on seven application benchmarks. All runtimes are in seconds and were calculated by taking the median 12-core execution time of 5 runs on an Intel Xeon X5650 with hyperthreading disabled. The runtimes of Prism and Prism-R include the time used to color the input graph. PR/G and PR/L run a PageRank algorithm on the web-Google [223] and soc-LiveJournal [14] graphs, respectively. ID/2000 and ID/4000 run an image denoise algorithm to remove Gaussian noise from 2D grayscale images of dimension 2000 by 2000 and 4000 by 4000. FBP/C1 and FBP/C3 perform belief propagation on a factor graph provided by the cora-1 and cora-3 datasets [322, 248]. ALS/N runs an alternating least squares algorithm on the NPIC-500 dataset [259].

graph itself with undirected edges. The idea behind chromatic scheduling is fairly simple. Chromatic scheduling begins by computing a ***(vertex) coloring*** of the conflict graph — an assignment of colors to the vertices such that no two adjacent vertices share the same color. Since no edge in the conflict graph connects two vertices of the same color, updates on all vertices of a given color can execute in parallel without producing races. To execute a round of a data-graph computation, the set of activated vertices Q is partitioned into χ ***color sets*** — subsets of Q containing vertices of a single color. Updates are applied to vertices in Q by serially stepping through each color set and updating all vertices within a color set in parallel. Indeed, the special case where the active set $Q == V$ is the entire graph (i.e., a static data-graph computation) can be executed using chromatic scheduling using Distributed GraphLab [233]. The result of a data-graph computation executed using chromatic scheduling is equivalent to that of a slightly modified version of Serial-DDGC that starts each round (immediately before line 9 of Figure 2-1) by sorting the vertices within its queue by color.

Chromatic scheduling avoids both of the pitfalls of the locking strategy. First, since only nonadjacent vertices in the conflict graph are updated in parallel, no races can occur, and the necessity for locks and their associated performance overheads are precluded. Second, by establishing a fixed order for processing different colors, any two adjacent vertices are always processed in the same order. The data-graph computation is therefore executed deterministically, as long as a deterministic coloring algorithm is used to color the conflict graph. While chromatic scheduling potentially loses parallelism because colors are processed serially, we shall see that this concern does not appear to be an issue in practice.

To date, chromatic scheduling has been applied to static data-graph computations [233], but not to dynamic data-graph computations. This chapter addresses the question of how to perform chromatic scheduling efficiently when the activation set changes on the fly, necessitating a data structure for maintaining dynamic sets of vertices in parallel.

Contributions

This chapter introduces PRISM, a chromatic-scheduling algorithm that executes dynamic data-graph computations in parallel efficiently in a deterministic fashion. PRISM employs a "multibag" data structure to manage an activation set as a list of color sets. The multibag achieves efficiency using "worker-local storage," which is memory locally associated with each "worker" thread executing the computation. By using the "multibag" and a deterministic coloring algorithm, PRISM guarantees to execute the data-graph computation deterministically.

We analyze the performance of PRISM using work-span analysis [77, Ch. 27]. The ***work*** of a computation is the total number of instructions executed, and the ***span*** corresponds to the longest path of dependencies in the parallel program. We shall make the reasonable assumption that a single update $f(v)$ executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span.[2] Under this assumption, on a degree-Δ data graph G colored using χ colors, PRISM executes the updates on the vertices in the activation set Q_r of a round r on P processors in $O(size(Q_r) + P)$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta) + \lg P)$ span.

The "price of determinism" incurred by using chromatic scheduling instead of the more common locking strategy appears to be negative for real-world applications. This discovery is perhaps surprising since it would seem to be strictly harder to guarantee that the computation behave deterministically than to allow for nondeterministic behaviors. Nevertheless, as Figure 2-2 indicates, on seven application benchmarks, PRISM executes 1.2–2.1 times faster than GraphLab's comparable, but nondeterministic, locking strategy, which we call RRLOCKS. This performance gap is not due solely to superior engineering or load balancing. A similar performance overhead is observed in a comparably engineered lock-based scheduling algorithm, CILK+LOCKS. PRISM outperforms CILK+LOCKS on each of the 7 application benchmarks and is on average (geometric mean) 1.4 times faster.

Our contribution is not a full-featured data-graph computation framework like GraphLab, Pregel, Galois, PowerGraph, Ligra, or GraphChi. Each of these systems is the result of countless hours of performance engineering and feature support. Instead, we provide a scheduling technique that could be adopted by any such framework to enable the deterministic execution of work-efficient, dynamic data-graph computations, which no existing framework currently supports[3] We use a modified shared-memory version of GraphLab in order to isolate the effect of our scheduling algorithms. Thus, the empirical comparisons in this chapter are apples-to-apples comparisons of scheduling strategies, not competitive comparisons with other systems.

PRISM behaves deterministically as long as every update is ***pure***: it modifies no data except for that associated with its target vertex. This assumption precludes the update function from modifying global variables to aggregate or collect values. To support this common use pattern, we describe an extension to PRISM, called PRISM-R, which executes dynamic data-graph computations deterministically even when updates modify global variables using associative operations. PRISM-R replaces each multibag PRISM uses with a "multivector," maintaining color sets whose contents are ordered deterministically. PRISM-R executes in the same theoretical bounds as PRISM, but its implementation is more involved. Empir-

[2]Other assumptions about the work and span of an update can easily be made at the potential expense of complicating the analysis.

[3]Deterministic Galois [268] has added support for deterministic execution of dynamic data-graph computations by recursively removing and executing independent sets of vertices. However, their algorithm is not work-efficient and, as a result, is much slower than the nondeterministic version.

ically Prism-R is on average (geometric mean) only 1.07 times slower than Prism and outperforms Cilk+Locks on each of the seven application benchmarks.

Outline

The remainder of this chapter is organized as follows. Section 2.2 reviews dynamic multithreading, the parallel programming model in which we describe and analyze our algorithms. Section 2.3 describes Prism, the chromatic-scheduling algorithm for dynamic data-graph computations. Section 2.4 describes the multibag data structure Prism uses to represent its color sets. Section 2.5 presents our theoretical analysis of Prism. Section 2.6 describes a Cilk Plus [170] implementation of Prism and presents empirical results measuring this implementation's performance on seven application benchmarks. Section 2.7 describes Prism-R which executes dynamic data-graph computations deterministically even when update functions modify global variables using associative operations. Section 2.8 describes and analyzes the multivector data structure Prism-R uses to represent deterministically ordered color sets. Section 2.9 analyzes Prism-R both theoretically, using work-span analysis, and empirically. Section 2.10 offers some concluding remarks.

2.2 Background

We implemented the Prism algorithm in Cilk Plus [170], a dynamic multithreading concurrency platform. This section provides background on the dag model of multithreading that embodies this and other similar concurrency platforms, including MIT Cilk [116], Cilk++ [219], Fortress [3], Habenero [16, 61], Hood [42], Java Fork/Join Framework [208], Task Parallel Library (TPL) [218], Threading Building Blocks (TBB) [291], and X10 [68]. We review the dag model of multithreading, the notions of work and span, and the basic properties of the work-stealing runtime systems underlying these concurrency platforms. We briefly discuss worker-local storage, which Prism's multibag data structure uses to achieve efficiency.

The dag model of multithreading

The dag model of multithreading [40, 41] is described in tutorial fashion in [77, Ch. 27]. The model views the executed computation resulting from running a parallel program as a ***computation dag*** in which each vertex denotes an instruction, and edges denote parallel control dependencies between instructions. To analyze the theoretical performance of a multithreaded program, such as Prism, we assume that the program executes on an ***ideal parallel computer***, where each instruction executes in unit time, the computer has ample bandwidth to shared memory, and concurrent reads and writes incur no overheads due to contention.

We shall assume that algorithms for the dag model are expressed using the keywords [77, Ch. 27] **spawn**, **sync**, and **parallel for**. The keyword **spawn** when preceding a function call F allows F to execute in parallel with its ***continuation*** — the statement immediately after the spawn of F. The complement of **spawn** is the keyword **sync**, which acts as a local barrier and prevents statements after the **sync** from executing until all earlier spawned functions return. These keywords can be used to implement other convenient parallel control constructs, such as the **parallel for** loop, which allows all of its iterations to operate logically in parallel. The work of a **parallel for** loop with n iterations is the total number of instructions in all executed iterations. The span is $\Theta(\lg n)$ plus the maximum span of any loop iteration. The $\Theta(\lg n)$ span term comes from the fact that the runtime system executes the loop iterations using parallel divide-and-conquer, and thus fans out the iterations as a balanced binary tree in the dag.

An important property of this model is notion of the ***serial elision*** of a program. The serial elision is the serial program that results when the keywords **spawn** and **sync** are elided and the **parallel for** is replaced by an ordinary **for** loop. The model guarantees that the serial elision of a program always provides a correct implementation of the program. That is, the keywords indicate opportunities for parallelism, but they do not require parallel execution. In this sense, every program in this model has a ***serial semantics***.

Work-span analysis

Given a multithreaded program whose execution is modeled as a dag A, we can bound the P-processor running time $T_P(A)$ of the program using ***work-span analysis*** [77, Ch. 27]. Recall that the work $T_1(A)$ is the number of instructions in A, and that the span $T_\infty(A)$ is the length of a longest path in A. Greedy schedulers [49, 99, 137] can execute a deterministic program with work T_1 and span T_∞ on P processors in time T_P satisfying

$$\max\{T_1/P, T_\infty\} \leq T_p \leq T_1/P + T_\infty \ , \tag{2.1}$$

and a similar bound can be achieved by more practical "work-stealing" schedulers [40, 41]. The ***speedup*** of an algorithm on P processors is T_1/T_P, which Inequality (2.1) shows to be at most P in theory. The ***parallelism*** T_1/T_∞ is the greatest theoretical speedup possible for any number of processors.

Work-stealing runtime systems

Runtime systems underlying concurrency platforms that support the dag model of multithreading usually implement a ***work stealing*** scheduler [57, 142, 41], which operates as follows. When the runtime system starts up, it allocates as many operating-system threads, called ***workers***, as there are processors. Each worker keeps a ***ready queue*** of tasks that can operate in parallel with the task it is currently executing. Whenever the execution of code generates parallel work, the worker puts the excess work into the queue. Whenever it needs work, it fetches work from its queue. When a worker's ready queue runs out of tasks, however, the worker becomes a ***thief*** and "steals" work from another ***victim*** worker's queue. If an application exhibits sufficient parallelism compared to the actual number of workers/processors, one can prove mathematically that the computation executes with linear speedup.

Worker-local storage

We refer to memory that is private to a particular worker thread as ***worker-local storage***. In a P-processor execution of a parallel program, a worker-local variable x can be implemented using a shared-memory array of length P. A worker accesses its local copy of x using a runtime-provided worker identifier to index the array of worker-local copies of x. The Cilk Plus runtime system, for example, provides the `__cilkrts_get_worker_number()` API call, which returns an integer identifying the current worker. Our implementation of

PRISM assumes the existence of a runtime-provided GET-WORKER-ID function that executes in $\Theta(1)$ time and returns an integer from 0 to $P-1$. Other strategies for implementing worker-local storage exist that are comparable to the strategy outlined here.

2.3 The Prism algorithm

This section presents PRISM, a chromatic-scheduling algorithm for executing dynamic data-graph computations deterministically. We describe how PRISM differs from the serial algorithm in Section 2.1, including how it maintains activation sets that are partitioned by color using the multibag data structure.

Figure 2-3 shows the pseudocode for PRISM, which differs from the SERIAL-DDGC routine from Figure 2-1 in two main ways: the use of a multibag data structure to implement Q, and the call to COLOR-GRAPH on line 1 to color the data graph.

A ***multibag*** Q represents a list $\langle C_0, C_1, \ldots, C_{\chi-1}\rangle$ of χ ***bags*** (unordered multisets) and supports two operations:

- MB-INSERT(Q, v, k) inserts an item v into bag C_k in Q. A multibag supports parallel MB-INSERT operations.
- MB-COLLECT(Q) produces a collection $\mathcal{C}$ that represents a list of the nonempty bags in Q, emptying Q in the process.

Although the multibag data structure supports duplicate items in a single bag, our implementation of PRISM actually ensures that no duplicate vertices are ever inserted into a bag.

PRISM calls COLOR-GRAPH on line 1 to color the given data graph $G = (V, E)$ and obtain the number χ of colors used. Although it is NP-complete to find an ***optimal*** coloring of a graph [119] — a coloring that uses the smallest possible number of colors — an optimal coloring is not necessary for PRISM to perform well, as long as the data graph is colored deterministically, in parallel,[4] and with sufficiently few colors in practice. Many parallel coloring algorithms exist that satisfy the needs of PRISM (see, for example, [5, 229, 179, 131, 146, 132, 330, 201, 200, 15]), however, our implementation of PRISM uses a multicore variant of the Jones and Plassmann algorithm [179] that produces a deterministic $(\Delta + 1)$-coloring of a degree-Δ graph $G = (V, E)$ in linear work and $O\Big(\lg V + \lg\Delta \cdot \min\{\sqrt{E}, \Delta + \lg\Delta \lg V/\lg\lg V\}\Big)$ span [146].

Let us now see how PRISM uses chromatic scheduling to execute a dynamic data-graph computation $\langle G, f, Q_0\rangle$. After line 1 colors G, line 3 initializes the multibag Q with the initial activation set Q_0, and then the **while** loop on lines 4–13 executes the rounds of the data-graph computation. At the start of each round, line 5 collects the nonempty bags $\mathcal{C}$ from Q, which correspond to the nonempty color sets for the round. Lines 6–12 iterate through the color sets $C \in \mathcal{C}$ sequentially, and the **parallel for** loop on lines 7–12 processes the vertices of each C in parallel. For each vertex $v \in C$, line 9 performs the update $S \leftarrow f(v)$, which returns a set S of activated vertices, and lines 10–12 insert into Q the vertices in S that have been activated.

Although a vertex u can be activated by multiple neighbors, it must only be updated at

[4] If the data-graph computation performs sufficiently many updates, a serial $\Theta(V + E)$-work greedy coloring algorithm, such as that introduced by Welsh and Powell [341], can suffice as well, since the time to color the graph would be sufficiently amortized against the work performed.

```
PRISM($G, f, Q_0$)
1  $\chi \leftarrow$ COLOR-GRAPH($G$)
2  $r \leftarrow 0$
3  $Q \leftarrow Q_0$
4  while $Q \neq \emptyset$
5      $\mathcal{C} \leftarrow$ MB-COLLECT($Q$)
6      for $C \in \mathcal{C}$
7          parallel for $v \in C$
8              $active[v] \leftarrow$ FALSE
9              $S \leftarrow f(v)$
10             parallel for $u \in S$
11                 if CAS($active[u]$, FALSE, TRUE)
12                     MB-INSERT($Q, u, color[u]$)
13         $r \leftarrow r + 1$

CAS($current, test, value$)
14 begin atomic
15 if $current == test$
16     $current \leftarrow value$
17     return TRUE
18 else
19     return FALSE
20 end atomic
```

Figure 2-3: Pseudocode for PRISM, including the compare-and-swap synchronization primitive CAS. The procedure PRISM takes as input a data graph G, an update function f, and an initial activation set Q_0. The procedure COLOR-GRAPH colors a given graph and returns the number of colors it used. The procedures MB-COLLECT and MB-INSERT operate the multibag Q to maintain activation sets for PRISM. The variable r tracks the number of rounds executed.

most once during a round. PRISM enforces this constraint[5] by using the atomic ***compare-and-swap*** operator [151, p. 480], which is available as a synchronization primitive on most machines and whose definition is given in lines 14–20. Lines 10–12 use the CAS primitive to activate each vertex $u \in S$ by atomically setting $active[u] \leftarrow$ TRUE, and if $active[u]$ was previously FALSE, then calling MB-INSERT. Thus, each vertex is inserted into Q at most once during a round.

Design considerations for the implementation of multibags

The theoretical performance of PRISM depends upon the properties of the multibag data structure. In particular, the multibag is carefully designed to ensure that PRISM is ***work-efficient*** — that is, it performs the same asymptotic work as the serial algorithm SERIAL-DDGC in Figure 2-1. Before examining the design of the multibag in Section 2.4, let us first explore why maintaining active color sets in PRISM in a work-efficient manner is tricky. Specifically, we shall consider two alternative strategies: bit vectors and an array of worker-local queues.

The bit-vector approach avoids the multibag altogether and simply manages activation sets using the bit vector *active* already used by PRISM. Recall that if $active[i]$ is TRUE, then the vertex $v_i \in V$ indexed by i is active. Suppose that *active* were the only data structure. To iterate over all activated vertices of color k, a **parallel for** could scan through *active*, updating the vertex v_i whenever $active[i]$ is TRUE and $color[i]$ is k. This scheme requires $\Omega(V\chi)$ work per round of the computation, where χ is the number of colors returned by COLOR-GRAPH in line 1 of Figure 2-3, since the entire bit vector must be scanned χ times each round. At the cost of additional preprocessing, *active* could be organized such that

[5]This constraint may be enforced without the use of an atomic compare-and-swap operation by deduplicating the contents of Q at the start of each round. However, our empirical studies have shown that this limited use of atomics is beneficial in practice.

vertices of the same color are assigned contiguous indexes. Even with this optimization, however, scanning *active* requires $\Omega(V)$ work each round, which is not work-efficient for dynamic computations that activate only a sparse subset of the vertices each round.

An alternative strategy that one might consider is to represent the active color sets using an array of worker-local queues. A straightforward implementation of this approach, however, is also not work-efficient. For a dynamic data-graph computation using χ colors and P processors, a total of $P\chi$ worker-local queues would be needed to maintain the set of active vertices, and $\Omega(P\chi)$ work would be required to collect all nonempty queues. As we shall see in Section 2.4, however, by using a carefully designed data structure to manage worker-local queues, we can obtain a work-efficient data structure for maintaining color sets.

2.4 The multibag data structure

This section presents the multibag data structure employed by PRISM. The multibag uses worker-local sparse accumulators [126] and an efficient parallel collection operation. We describe how the MB-INSERT and MB-COLLECT operations are implemented, and we analyze them using work-span analysis [77, Ch. 27]. When used in a P-processor execution of a parallel program, a multibag Q of χ bags storing n elements supports MB-INSERT in $\Theta(1)$ worst-case time and MB-COLLECT in $O(n+\chi+P)$ work and $O(\lg n+\chi+\lg P)$ span. Such a multibag storing k elements uses $O(P\chi+k)$ space.

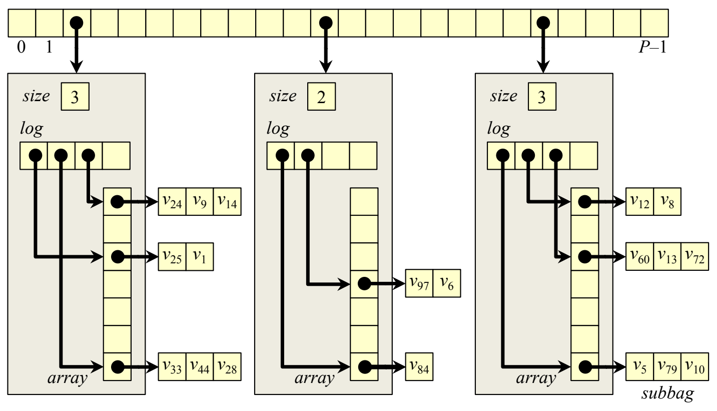
A ***sparse accumulator (SPA)*** [126] implements an array that supports lazy initialization of its elements. A SPA T contains a sparsely populated array $T.array$ of elements and a log $T.log$, which is a list of indices of initialized elements in $T.array$. To implement multibags, we shall only need the ability to create a SPA, access an arbitrary SPA element, or delete all elements from a SPA. For simplicity, we shall assume that an uninitialized array element in a SPA has a value of NIL. When an array element $T.array[i]$ is modified for the first time, the index i is appended to $T.log$. An appropriately designed SPA T storing $n = |T.log|$ elements admits the following performance properties:

- Creating T takes $\Theta(1)$ work.
- Each element of T can be accessed in $\Theta(1)$ work.
- Reading all k initialized elements of T takes $\Theta(k)$ work and $\Theta(\lg k)$ span.
- Emptying T takes $\Theta(1)$ work.

A multibag Q is an array of P worker-local SPA's, where P is the number of workers executing the program. We shall use p interchangeably to denote either a worker or that worker's unique identifier. Worker p's local SPA in Q is thus denoted by $Q[p]$. For a multibag Q of χ bags, each SPA $Q[p]$ contains an array $Q[p].array$ of size χ and a log $Q[p].log$. Figure 2-4(a) illustrates a multibag with $\chi = 7$ bags, 4 of which are nonempty. As Figure 2-4(a) shows, the worker-local SPA's in Q partition each bag $C_k \in Q$ into subbags $\{C_{k,0}, C_{k,1}, \ldots, C_{k,P-1}\}$, where $Q[p].array[k]$ stores subbag $C_{k,p}$.

Implementation of MB-Insert and MB-Collect

The worker-local SPA's enable a multibag Q to support parallel MB-INSERT operations without creating races. Figure 2-5 shows the pseudocode for MB-INSERT. When a worker p executes MB-INSERT(Q, v, k), it inserts element v into the subbag $C_{k,p}$ as follows. Line 1 calls GET-WORKER-ID to get worker p's identifier. Line 2 checks if subbag $C_{k,p}$ stored



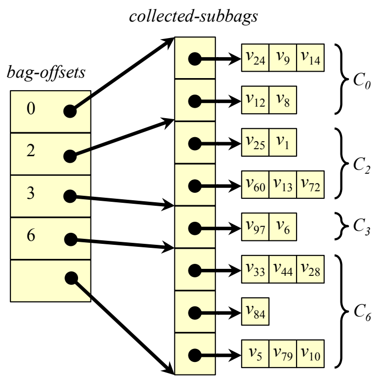


Figure 2-4: A multibag data structure. **(a)** A multibag containing 19 elements distributed across 4 distinct bags: $\{C_0, C_2, C_3, C_6\}$, representing vertices of colors 0, 2, 3, and 6, respectively. Each worker keeps track of its portion of a particular bag, its ***subbag***, using a worker-local SPA, thus avoiding initialization of unused subbags by maintaining a compact ***log*** pointing to the set of populated subbags. For example, bag C_6 is composed of three subbag contributions from the three active workers: $\{v_{33}, v_{44}, v_{28}\}$, $\{v_{84}\}$, and $\{v_5, v_{79}, v_{10}\}$. **(b)** The output of MB-Collect when executed on the multibag in **(a)**. Sets of subbags in *collected-subbags* are labeled with the bag C_k that their union represents.

```
MB-Insert($Q, v, k$)
1  $p \leftarrow$ Get-Worker-ID()
2  if $Q[p].array[k] ==$ NIL
3      Append($Q[p].log, k$)
4      $Q[p].array[k] \leftarrow$ new subbag
5  Append($Q[p].array[k], v$)
```

Figure 2-5: Pseudocode for the MB-Insert multibag operation. MB-Insert(Q, v, k) inserts the element v into the kth bag C_k of the multibag Q.

in $Q[p].array[k]$ is initialized, and if not, lines 3 and 4 initialize it. Line 5 inserts v into $Q[p].array[k]$.

Conceptually, the MB-Collect operation extracts the bags in Q to produce a compact representation of those bags that can be read efficiently. Figure 2-4(b) illustrates the compact representation of the elements of the multibag from Figure 2-4(a) that MB-Collect returns. This representation consists of a pair $\langle$*bag-offsets*, *collected-subbags*$\rangle$ of arrays that together resemble the representation of a graph in a CSR format. The *collected-subbags* array stores all of the subbags in Q sorted by their corresponding bag's index. The *bag-offsets* array stores indices in *collected-subbags* that denote the sets of subbags comprised by each bag. In particular, in this representation, the contents of bag C_k are stored in the subbags in *collected-subbags* between indices *bag-offsets*$[k]$ and *bag-offsets*$[k + 1]$.

Figure 2-6 sketches how MB-Collect converts a multibag Q stored in worker-local SPA's into the representation illustrated in Figure 2-4(b). Steps 1 and 2 create an array *collected-subbags* of nonempty subbags from the worker-local SPA's in Q. Each subbag $C_{k,p}$ in *collected-subbags* is tagged with the integer index k of its corresponding bag $C_k \in Q$. Step 3 sorts *collected-subbags* by these index tags, and Step 4 creates the *bag-offsets* array.

Step 5 removes all elements from Q, thereby emptying the multibag.

Analysis of multibags

We now analyze the work and span of the multibag's MB-INSERT and MB-COLLECT operations, starting with MB-INSERT.

Lemma 1 *Executing* MB-INSERT *takes* $\Theta(1)$ *time in the worst case.*

PROOF. Consider each step of a call to MB-INSERT(Q, v, k). The GET-WORKER-ID procedure on line 1 obtains the executing worker's identifier p from the runtime system in $\Theta(1)$ time, and line 2 checks if the entry $Q[p].array[k]$ is empty in $\Theta(1)$ time. Suppose that $Q[p].log$ and each subbag in $Q[p].array$ are implemented as dynamic arrays that use a deamortized table-doubling scheme [52]. Lines 3–5 then take $\Theta(1)$ time each to append k to $Q[p].log$, create a new subbag in $Q[p].array[k]$, and append v to $Q[p].array[k]$. □

The next lemma analyzes the work and span of MB-COLLECT.

Lemma 2 *In a P-processor parallel program execution, a call to* MB-COLLECT(Q) *on a multibag Q with χ bags whose contents are distributed across m distinct subbags executes in $O(m + \chi + P)$ work and $O(\lg m + \chi + \lg P)$ span.*

PROOF. We analyze each step of MB-COLLECT in turn. We shall use a helper procedure PREFIX-SUM(A), which computes the all-prefix sums of an array A of n integers in $\Theta(n)$ work and $\Theta(\lg n)$ span. (Blelloch [30] describes an appropriate implementation of PREFIX-SUM.) Step 1 replaces each entry in $Q[p].log$ in each worker-local SPA $Q[p]$ with the appropriate index-subbag pair $\langle k,\ C_{k,p}\rangle$ in parallel, which requires $\Theta(m + P)$ work and $\Theta(\lg m + \lg P)$ span. Step 2 gathers all index-subbag pairs into a single array. Suppose that each worker-local SPA $Q[p]$ is augmented with the size of $Q[p].log$, as Figure 2-4(a) illustrates. Executing PREFIX-SUM on these sizes and then copying the entries of $Q[p].log$ into *collected-subbags* in parallel therefore completes Step 2 in $\Theta(m+P)$ work and $\Theta(\lg m+\lg P)$ span. Step 3 can sort the *collected-subbags* array in $\Theta(m + \chi)$ work and $\Theta(\lg m + \chi)$ span using a variant of a parallel radix sort [74, 35, 355] as follows:

1. Divide *collected-subbags* into m/χ groups of size χ, and create an $(m/\chi) \times \chi$ matrix A, where entry A_{ij} stores the number of subbags with index j in group i. Constructing A can be done with $\Theta(m + \chi)$ work and $\Theta(\lg m + \chi)$ span by evaluating the groups in parallel and the subbags in each group serially.
2. Evaluate PREFIX-SUM on A^{T} (or, more precisely, the array formed by concatenating the columns of A in order) to produce a matrix B such that B_{ij} identifies which entries in the sorted version of *collected-subbags* will store the subbags with index j in group i. This PREFIX-SUM call takes $\Theta(m + \chi)$ work and $\Theta(\lg m + \lg \chi)$ span.
3. Create a temporary array T of size m, and in parallel over the groups of *collected-subbags*, serially move each subbag in the group to an appropriate index in T, as identified by B. Copying these subbags executes in $\Theta(m + \chi)$ work and $\Theta(\lg m + \chi)$ span.
4. Rename the temporary array T as *collected-subbags* in $\Theta(1)$ work and span.

Finally, Step 4 can scan *collected-subbags* for adjacent pairs of entries with different bag indices to compute *bag-offsets* in $\Theta(m)$ work and $\Theta(\lg m)$ span, and Step 5 can reset every SPA in Q in parallel using $\Theta(P)$ work and $\Theta(\lg P)$ span. Totaling the work and span of each step completes the proof. □

```
MB-COLLECT($Q$)
1. For each SPA $Q[p]$, map each bag index $k$ in $Q[p].log$ to the pair $\langle k,\ Q[p].array[k]\rangle$.
2. Concatenate the arrays $Q[p].log$ for all workers $p \in \{0, 1, \ldots, P-1\}$ into a single array,
   $collected\text{-}subbags$.
3. Sort the entries of $collected\text{-}subbags$ by their bag indices.
4. Create the array $bag\text{-}offsets$, where $bag\text{-}offsets[k]$ stores the index of the first subbag in
   $collected\text{-}subbags$ that contains elements of the $k$th bag.
5. For $p = 0, 1, \ldots, P-1$, delete all elements from the SPA $Q[p]$.
6. Return the pair $\langle bag\text{-}offsets,\ collected\text{-}subbags\rangle$.
```

Figure 2-6: Pseudocode for the MB-COLLECT multibag operation. Calling MB-COLLECT on a multibag Q produces a pair of arrays *collected-subbags*, which contains all nonempty subbags in Q sorted by their associated bag's index, and *bag-offsets*, which associates sets of subbags in Q with their corresponding bag.

Remark 3 *Let Q be a multibag in a P-processor execution with m distinct subbags that represents bags whose indices lie in the range $[0, k]$. Then Q may be treated as a multibag representing k bags so that* MB-COLLECT(Q) *executes in $O(m + k + P)$ work and $O(\lg m + k + \lg P)$ span.*

Although different executions of a program can store the elements of Q in different numbers m of distinct subbags, notice that m is never more than the total number of elements in Q.

2.5 Analysis of Prism

This section analyzes the performance of PRISM using work-span analysis [77, Ch. 27]. We derive bounds on the work and span of PRISM for any simple data-graph computation $\langle G, f, Q_0\rangle$. Recall that we make the reasonable assumptions that a single update $f(v)$ executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span, and that the update only activates vertices in $\text{Adj}[v]$. These work and span bounds can be used to characterize the data-graph computations on which PRISM achieves good parallel scalability. In particular, we show that on a data-graph on n vertices colored using χ colors that PRISM achieves good parallel speedup whenever the average work per round is much greater than $P\ \chi \lg n$

Let us first analyze the work and span of PRISM for one round of a data-graph computation.

Theorem 4 *Suppose that* PRISM *colors a degree-Δ data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0\rangle$. Then, on P processors,* PRISM *executes updates on all vertices in the activation set Q_r for a round r using $O(size(Q_r) + P)$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta) + \lg P)$ span.*

PROOF. Let us first analyze the work and span of one iteration of lines 6–12 in PRISM, which perform the updates on the vertices belonging to one color set $C \in Q_r$. Consider a vertex $v \in C$. Lines 8 and 9 execute in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span. For each vertex u in the set S of vertices activated by the update $f(v)$, Lemma 1 implies that lines 11–12 execute in $\Theta(1)$ total work. The **parallel for** loop on lines 10–12 therefore executes in $\Theta(S)$ work and $\Theta(\lg S)$ span. Because $|S| \le \deg(v)$, the **parallel for** loop on lines 7–12 thus executes in $\Theta(size(C))$ work and $\Theta(\lg C + \max_{v \in C} \lg(\deg(v))) = O(\lg C + \lg \Delta)$ span.

By processing each of the χ color sets belonging to Q_r, lines 6–12 therefore executes in $\Theta(\mathit{size}(Q_r) + \chi)$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta))$ span. Lemma 2 implies that line 5 executes MB-Collect in $O(Q_r + \chi_r + P)$ work and $O(\lg Q_r + \chi_r + \lg P)$ span where $\chi_r = \max_{v \in Q_r}\{\mathit{color}[v]\}$. Note that we take advantage here of the observation made in Remark 3. The theorem follows since $|Q_r| + \chi_r \leq \mathit{size}(Q_r) + 1$ □

Theoretical scalability of Prism

Dynamic data-graph computations typically run for multiple rounds until a convergence criteria is met. We will now generalize Theorem 4 to prove work and span bounds for Prism when executing a sequence of rounds.

Theorem 5 *Suppose that* Prism *colors a degree-Δ data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0 \rangle$ in r rounds applying updates to the activation sets $Q_0, Q_1, \ldots, Q_{r-1}$. Define the multiset $\mathcal{U} = \biguplus_{i=0}^{r-1} Q_i$ so that $|\mathcal{U}| = \sum_{i=0}^{r-1} |Q_i|$ and $\mathit{size}(\mathcal{U}) = \sum_{i=0}^{r-1} \mathit{size}(Q_i)$, where the symbol $\biguplus$ indicates a* ***multiset sum****.*[6] *Then, on P processors,* Prism *executes the data-graph computation using $O(\mathit{size}(\mathcal{U}) + rP)$ work and $O(r\ \chi(\lg((\mathcal{U}/r)/\chi) + \lg \Delta) + r \lg P)$ span.*

Proof. The work bound follows directly from Theorem 4 by taking the sum of work performed in each of the r rounds of Prism. The total span of Prism is equal to the sum of each round's span which by Theorem 4 is bounded by $\sum_{i=0}^{r-1}(\chi(\lg(Q_i/\chi) + \lg \Delta) + \lg P)$. Observing that $\sum_{i=0}^{r-1} \chi \lg(Q_i/\chi) \leq r\ \chi \lg((\mathcal{U}/r)/\chi)$ completes the proof. □

Given Theorem 5 we can compute the parallelism of Prism for a data-graph computation that applies a multiset $\mathcal{U}$ of updates over r rounds. The following corollary expresses the parallelism of Prism in terms of the average size of the activation sets in a sequence of rounds.

Corollary 6 *Suppose* Prism *executes a data-graph computation in r rounds during which it applies a multiset $\mathcal{U}$ of updates. Define the average number of updates per round $U_{avg} = |\mathcal{U}|/r$ and the average work per round $W_{avg} = \mathit{size}(\mathcal{U})/r$. Then* Prism *has $\Omega(W_{avg}/(\chi(\lg(U_{avg}/\chi) + \lg \Delta)))$ parallelism.*

Proof. Follows from Theorem 5 by computing the parallelism as the ratio of the work and span and then performing substitution. □

Corollary 6 implies that Prism achieves near perfect linear parallel speedup on P processors for a graph of n vertices when the average work performed in each round $W_{avg} \gg P\ \chi \lg n$.

2.6 Empirical evaluation

This section explores the performance properties of Prism from an empirical perspective. We describe three experiments designed to investigate the synchronization costs, dynamic-scheduling overheads, and scalability properties of Prism. For the first experiment, on a suite of 12 benchmark graphs, Prism executed between 1.0 and 2.1 times faster than a

[6] A multiset sum $\mathcal{M} = \biguplus_{i \in I} \mathcal{M}_i$ has multiplicity of element m equal to $\mathcal{M}(m) = \sum_{i \in I} \mathcal{M}_i(m)$ for all $m \in M$.

nondeterministic locking protocol on PageRank [51], exhibiting a geometric-mean speedup of a factor of 1.5, a substantial advantage in synchronization costs. The second experiment shows that the slowdown that PRISM incurs for dynamic scheduling using multibags, compared with static scheduling, is only about 1.16 when all vertices are activated in every round. This experiment shows that PRISM can be effective even for relatively densely activated graphs. The third experiment shows that PRISM scales well and is relatively insensitive to the number of colors needed to color the data graph, as long as there is sufficient parallelism.

Experimental setup

All of the benchmarks presented in this section were run on an Intel Xeon X5650 machine with 12 processor cores running at 2.67-GHz with hyperthreading disabled. Our test machine has 49 GB of DRAM, two 12-MB L3-caches, each shared among 6 cores, and private L2- and L1-caches of sizes 128 KB and 32 KB, respectively.

As a platform for our experiments, we implemented a new parallel execution engine within GraphLab [234] that uses Intel Cilk Plus[7] [170] to expose parallelism. The new execution engine and all of our scheduling algorithms were designed to be compatible with the original GraphLab API in order to facilitate a fair evaluation of the relative merits of different scheduling methodologies. In particular, to better understand the performance properties of PRISM, we developed four scheduling algorithms for comparison:

- SERIAL-DDGC is an implementation of the serial scheduling algorithm from Figure 2-1. SERIAL-DDGC provides a serial performance baseline for measuring the parallel speedup achieved by the other, more complex, scheduling algorithms for dynamic data-graph computations.
- CILK+LOCKS is a lock-based scheduling algorithm for dynamic data-graph computations. During each round, CILK+LOCKS updates only an active subset of the vertices in the graph. It uses a locking scheme to avoid executing conflicting updates in parallel. The locking scheme associates a shared-exclusive (i.e., reader-writer) lock [79] with each vertex in the graph. Prior to executing an update $f(v)$, vertex v's lock is acquired exclusively, and a shared lock is acquired for each $u \in \text{Adj}[v]$. A global ordering of locks is used to avoid deadlock.
- RRLOCKS is the lock-based dynamic scheduling algorithm implemented by the round-robin ***sweep scheduler*** in the original shared-memory version of GraphLab. A bit vector *active* is used to represent the active set of vertices. During each round, RRLOCKS scans each vertex in the active set in a round-robin fashion, conditionally updating a vertex v_i if *active*$[i]$ is TRUE. To avoid races, a locking strategy is used to coordinate updates that conflict.
- RRCOLOR is a coloring-based dynamic scheduling algorithm that uses a bit vector *active* to represent the active set of vertices. Instead of using locks to coordinate conflicting updates, however, RRCOLOR uses a vertex-coloring of the graph. At the start of the computation, RRCOLOR partitions the vertices by color and stores them in static arrays. For a graph colored using χ colors, each round of the computation is divided into χ color steps. During the kth color step, RRCOLOR scans all color-k vertices and conditionally updates a color-kd vertex v_i if *active*$[i]$ is TRUE.

[7]All code was compiled with Intel's ICC version 13.1.1.

Graph	$\|V\|$	$\|E\|$	χ	CILK+LOCKS	PRISM	PRISM-R	*Coloring*
cage15	5,154,860	94,044,700	17	36.9	35.5	35.6	12%
liveJournal	4,847,570	68,475,400	333	36.8	21.7	22.3	12%
randLocalDim25	1,000,000	49,992,400	36	26.7	14.4	14.6	18%
randLocalDim4	1,000,000	41,817,000	47	19.5	12.5	13.7	14%
rmat2Million	2,097,120	39,912,600	72	22.5	16.6	16.8	12%
powerGraph2M	2,000,000	29,108,100	15	12.1	9.8	10.1	13%
3dgrid5m	5,000,210	15,000,600	6	10.3	10.3	10.4	7%
2dgrid5m	4,999,700	9,999,390	4	17.7	8.9	9.0	4%
web-Google	916,428	5,105,040	43	3.9	2.4	2.4	8%
web-BerkStan	685,231	7,600,600	200	3.9	2.4	2.7	8%
web-Stanford	281,904	2,312,500	62	1.9	0.9	1.0	11%
web-NotreDame	325,729	1,469,680	154	1.1	0.8	0.8	12%

Figure 2-7: Performance of PRISM versus CILK+LOCKS when executing $10 \cdot |V|$ updates of the PageRank [51] data-graph computation on a suite of six real-world graphs and six synthetic graphs. Column "*Graph*" identifies the input graph, and columns $|V|$ and $|E|$ specify the number of vertices and edges in the graph, respectively. Column χ gives the number of colors PRISM used to color the graph. Columns "CILK+LOCKS," "PRISM," and "PRISM-R" present 12-core running times in seconds for each respective scheduler. Each running time is the median of 5 runs. Column "*Coloring*" gives the percentage of PRISM's running time spent coloring the graph. PRISM-R, discussed in Section 2.7, provides deterministic support for associative operations on global variables.

Overheads for locking and for chromatic scheduling

We compared the overheads associated with coordinating conflicting updates of a dynamic data-graph computation using locks versus using chromatic scheduling. We evaluated these overheads by comparing the 12-core execution times for PRISM and CILK+LOCKS to execute the PageRank [51] data-graph computation on a suite of graphs. We used PageRank for this study because of its comparatively cheap update function, which makes overheads due to scheduling more pronounced. PageRank updates a vertex v by first scanning v's incoming edges to aggregate the data from its incoming neighbors, and then by scanning v's outgoing edges to activate its outgoing neighbors.

We executed the PageRank application on a suite of six synthetic and six real-world graphs. The six real-world graphs came from the Stanford Large Network Dataset Collection (SNAP) [220], and the University of Florida Sparse Matrix Collection [84]. The six synthetic graphs were generated using the "randLocal," "powerLaw," "gridGraph," and "rMatGraph" generators included in the Problem Based Benchmark Suite [320]. We chose the graphs in this suite to be large enough to stress the memory system and thus make parallel speedup comparatively difficult. That is, given the random access inherent in data-graph computations, we expect most references to vertex data to come from DRAM, making DRAM bandwidth a scarce shared commodity. Since the span of PRISM is superconstant, however, for a fixed number of workers, increasing the size of the graph only increases parallelism, making good parallel speedup comparatively easy. Thus, we have pessimistically chosen the graphs in the suite to be large enough to make DRAM bandwidth a shared bottleneck but not unduly larger.

We observed that PRISM often performs slightly fewer rounds of updates than CILK+LOCKS when both are allowed to run until convergence. Wishing to isolate scheduling overheads, we controlled this variation by explicitly setting the total number of updates on a graph to

Benchmark	χ	*Updates*	RRLocks	RRColor	Prism	Prism-R
PR/L	333	48,475,700	35.25	14.5	17.7	18.4
ID/2000	4	40,000,000	63.15	50.1	59.2	59.9
FBP/C3	2	16,001,900	11.9	8.8	8.8	8.9
ID/1000	4	10,000,000	15.7	12.6	14.9	15.0
PR/G	43	9,164,280	3.1	1.3	2.1	2.2
FBP/C1	2	8,783,100	5.9	4.7	4.8	4.8
ALS/N	6	1,877,220	65.7	52.4	52.8	53.5

Figure 2-8: Performance of three schedulers on the seven application benchmarks from Figure 2-2, modified so that all vertices are activated in every round. Column "*Updates*" specifies the number of updates performed in the data-graph computation. Columns "RRLocks," "RRColor," "Prism," and "Prism-R" list the 12-core running times in seconds for the respective schedulers to execute each benchmark. Each running time is the median of 5 runs. The Prism-R algorithm, which provides deterministic support for associative operations on global variables, will be discussed in Section 2.7.

10 times the number of vertices.

Figure 2-7 presents the empirical results for this study. Figure 2-7 shows that over the 12 benchmark graphs, Prism executes between 1.0 and 2.1 times faster than Cilk+Locks on PageRank, exhibiting a geometric-mean speedup of a factor of 1.5. Moreover, from Figure 2-7 we see that an average of 10.9% of Prism's total running time is spent coloring the data graph, which is approximately equal to the cost of executing $|V|$ updates. Prism colors the data-graph once to execute the data-graph computation, however, meaning that its cost can be amortized over all of the updates in the data-graph computation. By contrast, the locking scheme implemented by Cilk+Locks incurs overhead for every update. Before updating a vertex v, Cilk+Locks acquires each lock associated with v and every vertex $u \in \text{Adj}[v]$. For simple data-graph computations whose update functions perform relatively little work, this step can account for a significant fraction of the time to execute an update.

Dynamic-scheduling overhead

To investigate the overhead of using multibags to maintain activation sets, we compared the 12-core running times of Prism, RRColor, and RRLocks on the seven benchmark applications from Figure 2-2. For this study, we modified the benchmarks slightly for each scheduler in order to provide a fair comparison. In particular, because Prism typically executes fewer updates than a static data-graph computation scheduler, we modified the update functions for each application so that every update on a vertex v always activates all vertices $u \in \text{Adj}[v]$. This modification guarantees that Prism executes the same set of updates each round as RRLocks and RRColor, while still incurring the overhead that Prism requires in order to maintain a dynamic set of active vertices. Thus, we compare the worst case conditions for Prism with respect to scheduling overhead with the best case conditions for RRLocks and RRColor.

Figure 2-8 presents the results of these tests, revealing that the overhead Prism incurs to maintain its activation sets using a multibag. As can be seen from the figure, Prism is 1.0 to 1.6 times slower than RRColor on the benchmarks with a geometric-mean relative slowdown of 1.16. That is, for static data-graph computations, Prism incurs only an aggregate 16% slowdown through the use of a multibag, as opposed to the simple array used by RRColor, which suffices for static scheduling. The Prism algorithm, which can also support *dynamic* activation sets efficiently, incurred minimal overhead for the multibag data structure. PRISM outperformed RRLOCKS on all benchmarks, achieving a geometric-mean speedup of 30% due to RRLOCKS's lock overhead. Thus, PRISM incurs relatively little overhead by maintaining activation sets with multibags.

The relative overhead of RRCOLOR and PRISM depends on the percentage of vertices active during a given round. As a typical example, RRCOLOR is approximately 1.09 times faster than PRISM on the image denoise benchmark when 80% of the vertices are active each round, but is 1.11 times slower when 5% or less of the vertices are active each round. As part of an effort to incorporate the PRISM scheduling paradigm into an existing data-graph computation framework (e.g., GraphLab, Pregel etc.), one might consider using a heuristic to switch between the use of a bitvector and a multibag depending on the density of the activation set. A simple heuristic such as a fixed threshold on the relative density of the activation set[8] (e.g., 10% of the vertices) would likely suffice to maintain activation sets with good performance: if fewer than 10% of vertices are active, use a multibag, otherwise use a bitvector.

Scalability of Prism

To measure the scalability of PRISM, and CILK+LOCKS, we compared their 12-core runtimes to the serial reference implementation SERIAL-DDGC. Figure 2-9 shows the empirical 12-core speedups relative to SERIAL-DDGC of PRISM and CILK+LOCKS on seven application benchmarks. Data for PRISM-R is also included, which will be discussed in Section 2.9. In geometric mean, CILK+LOCKS achieved 5.73 times speedup, PRISM achieved 7.56 times speedup, and PRISM-R achieved 7.42 times speedup.

In order to study the effect of the number χ of colors used to color the application's data graph on the parallel scalability of PRISM, we measured the parallelism T_1/T_∞ and the 12-core speedup T_1/T_{12} of PRISM while executing the image-denoise application as we varied the number of colors used. The image-denoise application performs belief propagation to remove Gaussian noise added to a gray-scale image. The data graph for the image-denoise application is a two-dimensional grid in which each vertex represents a pixel, and there is an edge between any two adjacent pixels. The COLOR-GRAPH procedure invoked in line 1 of Figure 2-3 typically colors this data-graph with just 4 colors.

To perform this study, we artificially increased χ by repeatedly taking a random nonempty subset of the largest set of vertices with the same color and assigning a new color to those vertices. Using this technique, we ran the image-denoise application on a 500-by-500 pixel input image for values of χ between 4 and $250,000$, the last data point corresponding to a coloring that assigns all pixels distinct colors. Figure 2-10 plots the results of these tests. Although the parallelism of PRISM is inversely proportional to χ, PRISM's speedup on 12 cores is relatively insensitive to χ, as long as the parallelism is greater than about 120. This result is consistent with the rule of thumb that a program with at least $10P$ parallelism should achieve nearly perfect linear speedup on P processors [77, p. 783].

[8] A similar heuristic was shown to be effective in the graph computation library Ligra [318] for adaptively switching between "dense" and "sparse" representations of vertex subsets.

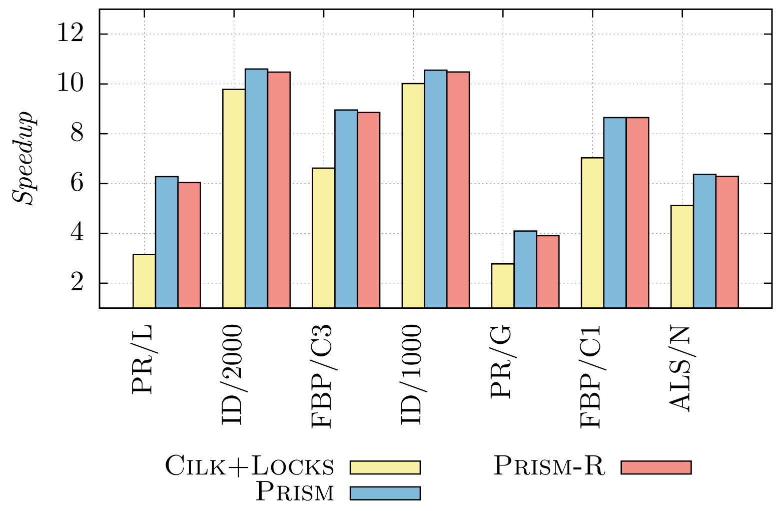


Figure 2-9: Empirical speedup relative to SERIAL-DDGC on 12 processor cores. Shown are the empirical speedups T_s/T_{12} of CILK+LOCKS, PRISM, and PRISM-R, where T_s is the runtime of the serial scheduling algorithm SERIAL-DDGC and T_{12} is the runtime of the particular algorithm on 12 cores. The PRISM-R algorithm is discussed in Section 2.7.

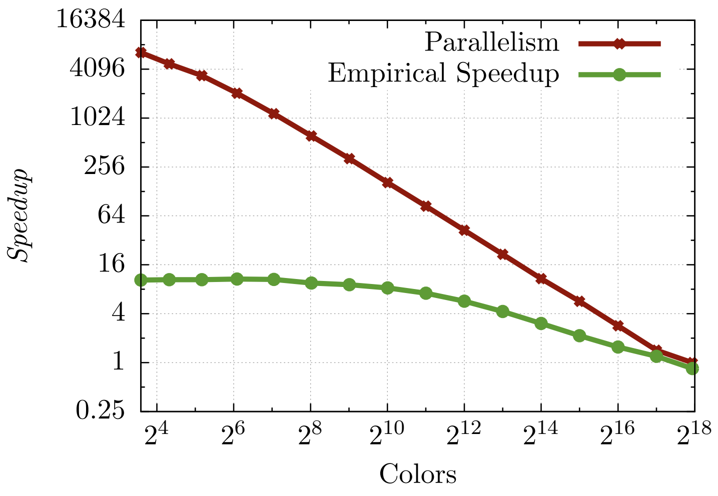


Figure 2-10: Scalability of PRISM on the image-denoise application as a function of χ, the number of colors used to color the data graph. The parallelism T_1/T_∞ is plotted together with the empirical speedup T_1/T_{12} achieved on a 12-core execution. Parallelism values were measured using the Cilkview scalability analyzer [149].

2.7 The Prism-R algorithm

This section introduces PRISM-R, a chromatic-scheduling algorithm that executes a dynamic data-graph computation deterministically even when updates modify global ***reducer variables*** using associative operations such as a reducer hyperobject [115]. While the chromatic scheduling technique employed by PRISM ensures that there are no data races on the vertex data of the graph, the order in which updates are made to a reducer variable among vertices of a common color can yield a nondeterministic result to the final reducer variable value. The multivector data structure, which is a theoretical improvement to the multibag, is used by PRISM-R to maintain activation sets that are partitioned by color and ordered deterministically. We describe an extension of the model of simple data-graph computations that permits an update function to perform associative operations on global variables using a parallel reduction mechanism. In this extended model, PRISM-R executes dynamic data-graph computations deterministically while achieving the same work and span bounds as PRISM.

Data-graph computations with global reductions

Several frameworks for executing data-graph computations allow updates to modify global variables in limited ways. Pregel aggregators [242], and GraphLab's sync mechanism [234], for example, both support data-graph computations in which an update can modify a global variable in a restricted manner. These mechanisms coordinate parallel modifications to a global variable using ***parallel reductions*** [175, 205, 31, 66, 197, 291, 169, 251], that is, they coordinate these modifications by applying them to local ***views*** (copies) of the variable and then ***reducing*** (combining) those copies together using a binary ***reduction operator***.

A ***reducer (hyperobject)*** [115, 213] is a general parallel reduction mechanism provided by Cilk Plus and other dialects of Cilk. A reducer is defined on an arbitrary data type T, called a ***view type***, by defining an IDENTITY operator and a binary REDUCE operator for views of type T. The IDENTITY operator creates a new view of the reducer. The binary REDUCE operator defines the reducer's reduction operator. A reducer is a particularly general reduction mechanism because it guarantees that, if its REDUCE operator is associative, then the final result in the global variable is deterministic: every parallel execution of the program produces the same result. Other parallel reduction mechanisms, including Pregel aggregators and GraphLab's sync mechanism, provide this guarantee only if the reduction operator is also commutative.

Although PRISM is implemented in Cilk Plus, PRISM does not produce a deterministic result if updates modify global variables using a noncommutative reducer. The reason for this is, in part, that the order of vertices within in a multibag depends on how the computation was scheduled among participating workers. As a result, the order in which lines 7–12 of PRISM in Figure 2-3 evaluates the vertices in a color set C is nondeterministic. If two updates on vertices in C modify the same reducer, then the relative order of these modifications can differ between runs of PRISM, even if a single worker happens to execute both updates.

PRISM-R

PRISM-R is an extension to PRISM that executes dynamic data-graph computations deterministically even when update functions are allowed to perform associative operations on global variables. The semantics of PRISM-R mimic that of SERIAL-DDGC when its

```
PRISM-R($G, f, Q_0$)
$\chi \leftarrow$ COLOR-GRAPH($G$)
$r \leftarrow 0$
$updates \leftarrow 0$
$Q \leftarrow Q_0$
while $Q \neq \emptyset$
    $\mathcal{C} \leftarrow$ MV-COLLECT($Q$)
    for $C \in \mathcal{C}$
        parallel for $i \leftarrow 1, 2, \ldots, |C|$
            $\langle v, p \rangle \leftarrow C[i]$
            if $p == priority[v]$
                $rank[f(v)] = updates + i$
                $priority[v] \leftarrow \infty$
                $S \leftarrow f(v)$
                parallel for $u \in S$
                    if PRIORITYWRITE($priority[u], rank[f(v)]$)
                        MV-INSERT($Q, \langle u, rank[f(v)] \rangle, color[u]$)
            $updates \leftarrow updates + |C|$
        $r \leftarrow r + 1$
```

```
PRIORITYWRITE($current, value$)
begin atomic
if $current > value$
    $current \leftarrow value$
    return TRUE
else
    return FALSE
end atomic
```

Figure 2-11: Pseudocode for PRISM-R. The algorithm takes as input a data graph G, an update function f, and an initial activation set Q_0. COLOR-GRAPH colors a given graph and returns the number of colors it used. The procedures MV-COLLECT and MV-INSERT operate the multivector Q to maintain activation sets for PRISM-R. PRISM-R updates the value of *updates* after processing each color set and r after each round of the data-graph computation.

queue of active vertices is stable sorted by color at the start of each round. In this modified version of SERIAL-DDGC updates to active vertices of the same color are applied in increasing order of their insertion into the queue. PRISM-R guarantees that the result of associative reductions performed by update functions reflect this same order.

Figure 2-11 shows the pseudocode for PRISM-R which differs from PRISM in its use of alternate data structure to maintain partitioned activation sets and in its use of a priority deduplication strategy for avoiding multiple updates to the same vertex in a round.

A ***multivector*** is used by PRISM-R to represent a list of χ ***vectors*** (ordered multisets). It supports the operations MV-INSERT and MV-COLLECT, which are analogous to the multibag operations MB-INSERT and MB-COLLECT, respectively. Each vector maintained by a multivector has serial semantics, meaning that the order of elements within each vector is deterministic and equivalent to the insertion order in an execution of the serial elision of the parallel program. Section 2.8 describes and analyzes the implementation of the multivector data structure.

The serial semantics of the multivector are not alone sufficient to ensure that updates are ordered deterministically in an execution of the serial elision of the program. Consider, for example, a round of PRISM that updates the three vertices x, y, z in parallel. Suppose that y activates u and both x and z activate a common neighbor v. The atomic compare-and-swap operator used by PRISM on line 11 of Figure 2-3 ensures that x and z will not both insert v into the activation set, but which of the two succeeds is nondeterministic. Inserting these two activated vertices into a multivector would produce either the order u, v or v, u depending on whether x or z activated v.

To eliminate this source of nondeterminism, PRISM-R assigns each update $f(v)$ a unique integer $rank[f(v)]$ on line 11 of Figure 2-11 that orders updates applied during a round according to their execution order in an execution of the serial elision of PRISM-R. Instead of maintaining a bit vector denoting whether or not a vertex is active PRISM-R maintains an integer array *priority* of priorities. For each active vertex v the value $priority[v]$ is equal to the smallest rank of any update $f(u)$ that activated v in the previous round. The priority of a vertex v is reset on line 12 before applying $f(v)$ by setting $priority[v] = \infty$.

For each vertex $u \in \text{Adj}[v]$ activated by update $f(v)$, PRISM-R uses an atomic ***priority-write*** operator [319] to set $priority[u] = \min\{priority[u], rank[f(v)]\}$ and inserts the vertex-priority pair $\langle u, rank[f(v)]\rangle$ into the multivector if the priority write is successful on line 15. The color sets returned by MV-COLLECT on line 6 can contain multiple vertex-priority pairs for each active vertex. On lines 8–16 PRISM-R iterates over the vertex-priority pairs $\langle v, p\rangle$ in a color set and only applies the update $f(v)$ if $priority[v] == p$. Since $priority[v]$ is equal to the lowest ranked update that activated v, PRISM-R updates each active vertex exactly once during a round in the same order as a serial execution.

2.8 The multivector data structure

This section introduces the multivector data structure, which provides a theoretical improvement to the multibag. The multivector data structure maintains several vectors (dynamic arrays), each supporting a parallel append operation. Each vector has serial semantics, that is, the order of elements within any vector is equivalent to their insertion order in an execution of the serial elision of the Cilk parallel program. The multivector can be used in place of the multibag to provide a stronger encapsulation of nondeterminism in programs whose behavior depends on the ordering of elements in each set. This section assumes familiarity with the Cilk execution model [116], as well as its implementation of reducers [115].

A ***multivector*** represents a list of χ ***vectors*** (ordered multisets). It supports the operations MV-INSERT and MV-COLLECT, which are analogous to the multibag operations MB-INSERT and MB-COLLECT, respectively. Our implementation relies on properties of a work-stealing runtime system. Consider a parallel program modeled by a computation dag A in the Cilk model of multithreading. The ***serial execution order*** $R(A)$ of the program lists the vertices of A according to the order they would be visited if an execution of the serial elision of the underlying Cilk program were executed, which corresponds to a left-to-right depth-first execution of the dag.

A work-stealing scheduler partitions $R(A)$ into a sequence $R(A) = \langle t_0, t_1, \ldots, t_{M-1}\rangle$, where each ***trace*** $t_i \in R(A)$ is a contiguous subsequence of $R(A)$ executed by exactly one worker. A multivector represents each vector as a sequence of ***trace-local subvectors*** — subvectors that are modified within exactly one trace. The ordering properties of traces imply that concatenating a vector's trace-local subvectors in order produces a vector whose elements appear in the serial execution order. The multivector data structure assumes that a worker can query the runtime system to determine when it starts executing a new trace.

The log-tree reducer

A multivector stores its nonempty trace-local subvectors in a ***log tree***, which represents an ordered multiset of elements and supports $\Theta(1)$-work append operations. A log tree is a binary tree in which each node L stores a dynamic array $L.sublog$. The ordered multiset that

```
FLATTEN($L, A, i$)
  $A[i] \leftarrow L$
  if $L.left \neq$ NIL
      spawn FLATTEN($L.left, A, i - L.right.size - 1$)
  if $L.right \neq$ NIL
      FLATTEN($L.right, A, i - 1$)
  sync
```

Figure 2-12: Pseudocode for the FLATTEN operation for a log tree. FLATTEN performs a post-order parallel traversal of a log tree to place its nodes into a contiguous array.

```
IDENTITY()
  $L \leftarrow$ new log-tree node
  $L.sublog \leftarrow$ new vector
  $L.size \leftarrow 1$
  $L.left \leftarrow$ NIL
  $L.right \leftarrow$ NIL
  return $L$
```

```
REDUCE($L_l, L_r$)
  $L \leftarrow$ IDENTITY()
  $L.size \leftarrow L_l.size + L_r.size + 1$
  $L.left \leftarrow L_l$
  $L.right \leftarrow L_r$
  return $L$
```

Figure 2-13: Pseudocode for the IDENTITY and REDUCE log-tree reducer operations. The IDENTITY operation creates and returns a new log-tree node L. The REDUCE(L_l, L_r) operation concatenates a left log-tree node L_l with a right log-tree node L_r.

a log tree represents corresponds to a concatenation of the tree's dynamic arrays in a post-order tree traversal. Each log-tree node L is augmented with the size of its subtree $L.size$ counting the number of log-tree nodes in the subtree rooted at L. Using this augmentation, the operation FLATTEN($L, A, L.size - 1$) described in Figure 2-12 flattens a log tree rooted at L of n nodes and height h into a contiguous array A using $\Theta(n)$ work and $\Theta(h)$ span.

To handle parallel MV-INSERT operations, a multivector employs a ***log-tree reducer***, that is, a Cilk Plus reducer whose view type is a log tree. Figure 2-13 presents the pseudocode for the IDENTITY and REDUCE operations for the log-tree reducer.

The IDENTITY operation creates a new log-tree node with an empty sublog. The REDUCE(L_l, L_r) operation creates a new root node L and assigns $L.left = L_l$ and $L.right = L_r$. Updates are performed using a log-tree reducer R by first obtaining a local view L of the log-tree reducer using a runtime-provided function GET-LOCAL-VIEW(R) and appending elements to $L.sublog$. A log tree's FLATTEN operation uses a post-order traversal to order the log tree's nodes, which results in an ordering identical to that which would be obtained by using a linked-list reducer in place of the log-tree reducer.

The log-tree reducer's REDUCE operation is logically associative, that is, for any three log-tree reducer views a, b, and c, the views produced by REDUCE(REDUCE(a, b), c) and REDUCE(a, REDUCE(b, c)) represent the same ordered multiset.

Figure 2-14 illustrates the state of a log-tree reducer R following the execution of a fork-join parallel function $A(R)$. Steals occur on line 2 of A and line 8 of B. The log-tree reducer partitions this execution of $A(R)$ into 5 traces each of which corresponds to one node in the tree. The first trace corresponds to the worker that begins the execution of $A(R)$ and each steal creates two additional traces: one corresponding to the stolen continuation of the spawned function, and another corresponding to the portion of the program following the associated **sync** statement.

To maintain trace-local subvectors, a multivector Q consists of an array of P worker-

```
A(R)
1   Log-Insert($R, e_1$)
2   spawn B($R$)
3   Log-Insert($R, e_7$)
4   sync
5   Log-Insert($R, e_8$)

B(R)
 6  Log-Insert($R, e_2$)
 7  spawn Log-Insert($R, e_3$)
 8  Log-Insert($R, e_4$)
 9  Log-Insert($R, e_5$)
10  sync
11  Log-Insert($R, e_6$)

Log-Insert(R, e)
12  $L \leftarrow$ Get-Local-View($R$)
13  Append($L.subblog, e$)
```

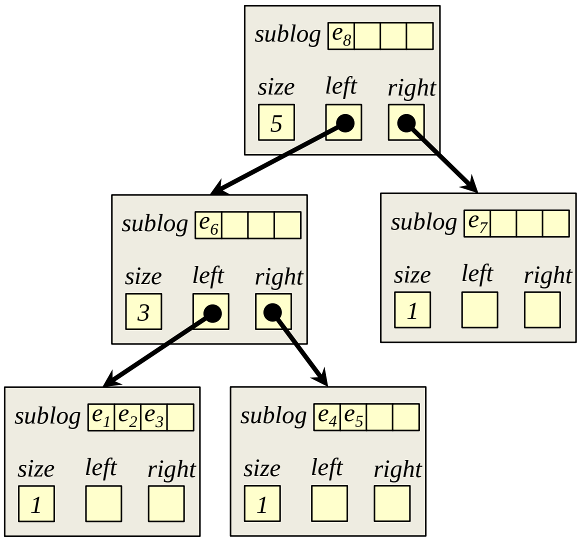


Figure 2-14: The state of a log-tree reducer R after a work-stealing execution of A(R). Steals occur on line 2 of A and line 8 of B partitioning the execution into 5 traces. The ordered multiset $(e_1, e_2, \ldots, e_8)$ is represented by 5 trace-local sublogs ordered according to a post-order traversal of the log tree.

local SPA's, where P is the number of processors executing the computation, and a log-tree reducer. The SPA $Q[p]$ for worker p stores the trace-local subvectors that worker p has appended since the start of its current trace. The log-tree reducer $Q.log\text{-}reducer$ stores all nonempty subvectors created.

Let us see how MV-Insert and MV-Collect are implemented.

Figure 2-16 sketches the MV-Insert(Q, v, k) operation to insert element v into the vector $C_k \in Q$. MV-Insert differs from MB-Insert in two ways. First, when a new subvector is created and added to a SPA, lines 19–20 additionally append that subvector to $Q.log\text{-}reducer$, thereby maintaining the log-tree reducer. Second, lines 15–16 reset the contents of the SPA $Q[p]$ after worker p begins executing a new trace, thereby ensuring that $Q[p]$ stores only trace-local subvectors.

Figure 2-15 sketches the MV-Collect operation. The return value of MV-Collect is a pair ⟨*subvector-offsets*, *collected-subvectors*⟩ analogous to the return value of MB-Collect. The procedure MV-Collect differs from MB-Collect primarily in that Step 1, which replaces Steps 1 and 2 in MB-Collect, flattens the log tree underlying $Q.log\text{-}reducer$ to produce the unsorted array *collected-subvectors*. MV-Collect also requires that *collected-subvectors* be sorted using a stable sort on Step 2. The integer sort described in the proof of Lemma 2 for MB-Collect is a suitable stable sort for this purpose.

Analysis of multivector operations

We now analyze the work and span of the MV-Insert and MV-Collect operations, starting with MV-Insert.

```
MV-COLLECT($Q$)
  1. Flatten the log-reducer tree so that all subvectors in the log appear in a contiguous array
     $collected\text{-}subvectors$.
  2. Sort the subvectors in $collected\text{-}subvectors$ by their vector indices using a stable sort.
  3. Create the array $vector\text{-}offsets$, where $vector\text{-}offsets[k]$ stores the index of the first subvector
     in $collected\text{-}subvectors$ that contains elements of the vector $C_k \in Q$.
  4. Reset $Q.log\text{-}reducer$, and for $p = 0, 1, \ldots, P-1$, reset $Q[p]$.
  5. Return the pair $\langle vector\text{-}offsets,\ collected\text{-}subvectors \rangle$.
```

Figure 2-15: Pseudocode for the MV-COLLECT multivector operation. Calling MV-COLLECT on a multivector Q produces a pair $\langle vector\text{-}offsets,\ collected\text{-}subvectors \rangle$ of arrays, where *collected-subvectors* contains all nonempty subvectors in Q sorted by their associated vector's color, and *vector-offsets* associates sets of subvectors in Q with their corresponding vector.

```
MV-INSERT($Q, v, k$)
14  $p \leftarrow$ GET-WORKER-ID()
15  if worker $p$ began a new trace since last insert
16      reset $Q[p]$
17  if $Q[p].array[k] ==$ NIL
18      $Q[p].array[k] \leftarrow$ new subvector
19      $L \leftarrow$ GET-LOCAL-VIEW($Q.log\text{-}reducer$)
20      APPEND($L.sublog, Q[p].array[k]$)
21  APPEND($Q[p].array[k], v$)
```

Figure 2-16: Pseudocode for the MV-INSERT multivector operation. MV-INSERT(Q, v, k) inserts an element v into the kth vector C_k maintained by the multivector Q.

Lemma 7 *Executing* MV-INSERT *takes* $\Theta(1)$ *time in the worst case.*

PROOF. Resetting the SPA $Q[p]$ on line 16 can be done in $\Theta(1)$ worst-case time with an appropriate SPA implementation, and appending a new subvector to a log tree takes $\Theta(1)$ time. The theorem thus follows from the analysis of MB-INSERT in Lemma 1. □

Lemma 8 bounds the work and span of MV-COLLECT.

Lemma 8 *Consider a computation A with span $T_\infty(A)$, and suppose that the contents of a multivector Q of χ vectors are distributed across m subvectors. Then a call to* MV-COLLECT(Q) *incurs $\Theta(m + \chi)$ work and $\Theta(\lg m + \chi + T_\infty(A))$ span.*

PROOF. Flattening the log-tree reducer in Step 1 is accomplished in two steps. First, the FLATTEN operation writes the nodes of the log tree to a contiguous array. Execution of FLATTEN has span proportional to the depth of the log tree, which is bounded by $O(T_\infty(A))$, since at most $O(T_\infty(A))$ reduction operations can occur along any path in A, and REDUCE for log trees executes in $\Theta(1)$ work [115]. Second, using a parallel-prefix sum computation, the log entries associated with each node in the log tree can be packed into a contiguous array, incurring $\Theta(m)$ work and $\Theta(\lg m)$ span. Step 1 thus incurs $\Theta(m)$ work and $O(\lg m + T_\infty(A))$ span. The remaining steps of MV-COLLECT, which are analogous to those of MB-COLLECT and analyzed in Lemma 2, execute in $\Theta(\chi + \lg m)$ span. □

2.9 Analysis and evaluation of Prism-R

This section presents a theoretical work-span analysis of PRISM-R, demonstrating that its work and span are asymptotically equivalent to PRISM. This section also discusses PRISM-R's empirical performance relative to PRISM, which was evaluated in Section 2.6. In particular, PRISM-R is only 2-7% slower than PRISM, overall, while providing deterministic support for associative operations on global variables.

Work-span analysis of Prism-R

We begin by analyzing the work and span of PRISM-R for simple data-graph computations that perform associative operations on global variables. In this extended model, PRISM-R executes dynamic data-graph computations deterministically while achieving the same work and span bounds as PRISM.

Theorem 9 *Let G be a degree-Δ data graph. Suppose that* PRISM-R *colors G using χ colors. Then* PRISM-R *executes updates on all vertices in the activation set Q_r for a round r of a simple data-graph computation $\langle G, f, Q_0\rangle$ in $O(size(Q_r))$ work and $O(\chi(\lg(Q_r/\chi) + \lg\Delta))$ span.*

PROOF. PRISM-R can perform a priority write to its *active* array with $\Theta(1)$ work, and it can remove duplicates from the output of MV-COLLECT in $O(size(Q_r))$ work and $O(\lg(size(Q_r))) = O(\lg Q_r + \lg\Delta)$ span. The theorem follows by applying Lemmas 7 and 8 appropriately to the analysis of PRISM in Theorem 4. □

Theorem 10 *Suppose that* PRISM-R *colors a degree-Δ data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0\rangle$ in r rounds applying updates to the activation sets $Q_0, Q_1, \ldots, Q_{r-1}$. Define the multiset $\mathcal{U} = \biguplus_{i=0}^{r-1} Q_i$ so that $|\mathcal{U}| = \sum_{i=0}^{r-1} |Q_i|$ and $size(\mathcal{U}) = \sum_{i=0}^{r-1} size(Q_i)$. Then* PRISM-R *executes the data-graph computation using $O(size(\mathcal{U}))$ work and $O(r \cdot \chi(\lg((\mathcal{U}/r)/\chi) + \lg\Delta))$ span.*

PROOF. By Theorem 9 PRISM-R executes a round of a data-graph computation using the same asymptotic work and span as PRISM. We mirror the arguments in Theorem 5 to bound the work and span of PRISM-R for a sequence of rounds. □

Given Theorem 10 we can compute the parallelism of PRISM-R for a data-graph computation that applies a multiset $\mathcal{U}$ of updates over r rounds. The following corollary expresses the parallelism of PRISM-R in terms of the average size of the activation sets in a sequence of rounds.

Corollary 11 *Suppose* PRISM-R *executes a data-graph computation in r rounds during which it applies a multiset $\mathcal{U}$ of updates. Define the average number of updates per round $U_{avg} = |\mathcal{U}|/r$ and the average work per round $W_{avg} = size(\mathcal{U})/r$. Then* PRISM-R *has $\Omega(W_{avg}/(\chi(\lg(U_{avg}/\chi) + \lg\Delta)))$ parallelism.*

PROOF. Follows from Theorem 10 by computing the parallelism as the ratio of the work and span and then performing substitution. □

Empirical evaluation of Prism-R

Prism-R provides deterministic support for associative operations on global variables at the cost of additional complexity versus Prism, specifically in the maintenance of activation sets. Nonetheless, Prism-R guarantees the same asymptotic work and span as Prism. Empirically, we find that Prism-R suffers a geomean slowdown of only 2-7% versus Prism in various scenarios. In particular, the 12-core performance for each dynamic data-graph computation application featured in Figure 2-2 demonstrate that for real-world applications Prism-R is 7% slower in geometric mean than Prism. in Figure 2-8 we see that Prism-R is only 1.8% slower than Prism for static versions of the applications featured in Figure 2-2 (i.e., all vertices are updated every round). Finally, in Figure 2-7 we present the 12-core performance of Prism-R on PageRank [51] for a suite of six synthetic and six real-world graphs. In this case, Prism-R is 3.5% slower in geometric mean than Prism.

2.10 Conclusion

Researchers over multiple decades have soberly advised the rest of the community that the difficulty of parallel programming can be greatly reduced by using some form of deterministic parallelism [281, 143, 125, 326, 32, 105, 92, 93, 160, 23, 24, 274, 354, 43]. With a deterministic parallel program, the programmer observes no logical concurrency, that is, no nondeterminacy in the behavior of the program due to the relative and nondeterministic timing of communicating processes (e.g., when two processes try to acquire a lock simultaneously). The semantics of a *deterministic* parallel program are therefore serial and reasoning about such a program's correctness is theoretically no harder than reasoning about the correctness of a serial program, which is already sufficiently hard for most people. Testing, debugging, and formal verification is simplified by determinism, because there is no need to consider all possible relative timings (i.e., interleavings) of operations on shared mutable state.

The behavior of Prism corresponds to a variant of Serial-DDGC that sorts the activated vertices in its queue by color at the start of each round. Whether Prism executes a given data graph on 1 processor or many, it always behaves the same way. With Prism-R, this property holds even when the update function can perform reductions (e.g., associative operators on global variables). By contrast, lock-based schedulers provide no such a guarantee of determinism. Instead, updates in a round executed by a lock-based scheduler appear to execute according to some linear order, the so-called *sequential consistency* model employed by GraphLab [234, 233] and others. This order is nondeterministic due to races on the acquisition of locks.

Blelloch, Fineman, Gibbons, and Shun [33] argue that deterministic programs can be fast compared with nondeterministic programs, and they document many examples where the overhead for converting a nondeterministic program into a deterministic one is small. They even document a few cases where this "price of determinism" is *slightly* negative. To their list, we add the execution of dynamic data-graph computations as having a price of determinism which is *significantly* negative.

2.11 Acknowledgments

Thanks to Guy Blelloch of Carnegie Mellon University for sharing utility functions from his Problem Based Benchmark Suite [320]. Thanks to Aydın Buluç of Lawrence Berkeley Laboratory for helping us in our search for collections of large sparse graphs. Thanks to Uzi Vishkin of University of Maryland for helping us track down early work on parallel sorting. Thanks to Fredrik Kjolstad, Angelina Lee, and Justin Zhang of MIT CSAIL and Guy Blelloch, Julian Shun, and Harsha Vardhan Simhadri of Carnegie Mellon University for providing helpful discussions.

Chapter 3

Ordering Heuristics for Parallel Graph Coloring

This chapter discusses efficient parallel algorithms for coloring graphs that are deterministic and have the semantics of the commonly used serial graph-coloring algorithm from Welsh and Powell. The serial semantics of the parallel algorithm described in this chapter are leveraged to design principled ordering heuristics for parallel graph coloring that can be viewed and analyzed as coarsened variants of existing heuristics used in serial codes. This work was conducted in collaboration with William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson.

Abstract

This chapter introduces the largest-log-degree-first (LLF) and smallest-log-degree-last (SLL) ordering heuristics for parallel greedy graph-coloring algorithms, which are inspired by the largest-degree-first (LF) and smallest-degree-last (SL) serial heuristics, respectively. We show that although LF and SL, in practice, generate colorings with relatively small numbers of colors, they are vulnerable to adversarial inputs for which any parallelization yields a poor parallel speedup. In contrast, LLF and SLL allow for provably good speedups on arbitrary inputs while, in practice, producing colorings of competitive quality to their serial analogs.

We applied LLF and SLL to the parallel greedy coloring algorithm introduced by Jones and Plassmann, referred to here as JP. Jones and Plassmann analyze the variant of JP that processes the vertices of a graph in a random order, and show that on an $O(1)$-degree graph $G = (V, E)$, this JP-R variant has an expected parallel running time of $O(\lg V / \lg\lg V)$ in a PRAM model. We improve this bound to show, using work-span analysis, that JP-R, augmented to handle arbitrary-degree graphs, colors a graph $G = (V, E)$ with degree Δ using $\Theta(V + E)$ work and $O(\lg V + \lg \Delta \cdot \min\{\sqrt{E}, \Delta + \lg \Delta \lg V / \lg\lg V\})$ expected span. We prove that JP-LLF and JP-SLL— JP using the LLF and SLL heuristics, respectively — execute with the same asymptotic work as JP-R and only logarithmically more span while producing higher-quality colorings than JP-R in practice.

We engineered an efficient implementation of JP for modern shared-memory multicore computers and evaluated its performance on a machine with 12 Intel Core-i7 (Nehalem) processor cores. Our implementation of JP-LLF achieves a geometric-mean speedup of 7.83 on eight real-world graphs and a geometric-mean speedup of 8.08 on ten synthetic graphs, while our implementation using SLL achieves a geometric-mean speedup of 5.36 on these real-world graphs and a geometric-mean speedup of 7.02 on these synthetic graphs. Fur-

thermore, on one processor, JP-LLF is slightly faster than a well-engineered serial greedy algorithm using LF, and likewise, JP-SLL is slightly faster than the greedy algorithm using SL.

3.1 Introduction

Graph coloring is a heavily studied problem with many real-world applications, including the scheduling of conflicting jobs [341, 244, 10, 109], register allocation [64, 63, 50], high-dimensional nearest-neighbor search [22], and sparse-matrix computation [180, 297, 75], to name just a few. Formally, a ***(vertex)-coloring*** of an undirected graph $G = (V, E)$ is an assignment of a ***color*** color(v) to each vertex $v \in V$ such that for every edge $(u, v) \in E$, we have color$(u) \neq$ color(v), that is, no two adjacent vertices have the same color. The ***graph-coloring problem*** is the problem of determining a coloring which uses as few colors as possible.

We were motivated to work on graph coloring in the context of "chromatic scheduling" [26, 1, 184] of parallel "data-graph computations." A ***data graph*** is a graph with data associated with its vertices and edges. A ***data-graph computation*** is an algorithm implemented as a sequence of "updates" on the vertices of a data graph $G = (V, E)$, where ***updating*** a vertex $v \in V$ involves computing a new value associated with v as a function of v's old value and the values associated with the ***neighbors*** of v: the set of vertices adjacent to v in G, denoted $\text{Adj}[v] = \{u \in V : (v, u) \in E\}$. To ensure atomicity of each update, rather than using mutual-exclusion locks or other nondeterministic means of data synchronization, chromatic scheduling first colors the vertices of G and then sequences through the colors, scheduling all vertices of the same color in parallel. The time to perform a data-graph computation thus depends both on how long it takes to color G and on the number of colors produced by the graph-coloring algorithm: more colors means less parallelism. Although the coloring can be performed offline for some data-graph computations, for other computations the coloring must be produced online, and one must accept a trade-off between coloring ***quality*** — number of colors — and the time to produce the coloring.

Although the problem of finding an ***optimal*** coloring of a graph — a coloring using the fewest colors possible — is NP-complete [120], heuristic "greedy" algorithms work reasonably well in practice. Welsh and Powell [341] introduced the original ***greedy*** coloring algorithm, which iterates over the vertices and assigns each vertex the smallest color not assigned to a neighbor. For a graph $G = (V, E)$, define the ***degree*** of a vertex $v \in V$ by $\deg(v) = |\text{Adj}[v]|$, the number of neighbors of v, and let the ***degree*** of G be $\Delta = \max_{v \in V}\{\deg(v)\}$. Welsh and Powell show that the greedy algorithm colors a graph G with degree Δ using at most $\Delta + 1$ colors.

Ordering heuristics

In practice, however, greedy coloring algorithms tend to produce much better colorings than the $\Delta+1$ bound implies, and moreover, the order in which a greedy coloring algorithm colors the vertices affects the quality of the coloring.[1] To reduce the number of colors a greedy coloring algorithm uses, practitioners therefore employ ***ordering heuristics*** to determine the order in which the algorithm colors the vertices [179, 4, 48, 246].

[1]In fact, for any graph $G = (V, E)$, some ordering of V causes a greedy algorithm to color G optimally, although finding such an ordering is NP-hard [260].

The literature includes many studies of ordering heuristics and how they affect running time and coloring quality. Here are six of the more popular heuristics:

FF The ***first-fit*** ordering heuristic [341, 232] colors vertices in the order they appear in the input graph representation.

R The ***random*** ordering heuristic [179] colors vertices in a uniformly random order.

LF The ***largest-degree-first*** ordering heuristic [341] colors vertices in order of decreasing degree.

ID The ***incidence-degree*** ordering heuristic [75] iteratively colors an uncolored vertex with the largest number of colored neighbors.

SL The ***smallest-degree-last*** ordering heuristic [246, 4] colors the vertices in the order induced by first removing all the lowest-degree vertices from the graph, then recursively coloring the resulting graph, and finally coloring the removed vertices.

SD The ***saturation-degree*** ordering heuristic [48] iteratively colors an uncolored vertex whose colored neighbors use the largest number of distinct colors.

The experimental results overviewed in the Appendix (Section 3.11) indicate that we have listed these heuristics in rough order of coloring quality from worst to best, confirming the findings of Gebremedhin and Manne [122], who also rank the relative quality of R, LF, ID, and SD in this order.

Although an ordering heuristic can be viewed as producing a permutation of the vertices of a graph $G = (V, E)$, we shall find it convenient to think of an ordering heuristic H as producing an injective (1-to-1) ***priority function*** $\rho : V \to \mathbb{R}$.[2] We shall use the notation $\rho \in H$ to mean that the ordering heuristic H produces a priority function ρ.

Figure 3-1 gives the pseudocode for GREEDY, a greedy coloring algorithm. GREEDY takes a vertex-weighted graph $G = (V, E, \rho)$ as input, where $\rho : V \to \mathbb{R}$ is a priority function produced by some ordering heuristic. Each step of GREEDY simply selects the uncolored vertex with the highest priority according to ρ and colors it with the smallest available color. Generally, for a coloring algorithm A and ordering heuristic H, let A-H denote the coloring algorithm A that runs on vertex-weighted graphs whose priority functions are produced by H. In this way, we separate the behavior of the coloring algorithm from that of the ordering heuristic.

GREEDY, using any of these six ordering heuristics, can be made to run in $\Theta(V + E)$ time theoretically. Although some of these ordering heuristics involve more bookkeeping than others, achieving these theoretical bounds for GREEDY-FF, GREEDY-R, GREEDY-LF, GREEDY-ID, and GREEDY-SL is straightforward [129, 246]. Despite conjectures to the contrary [129, 75], GREEDY-SD can also be made to run in $\Theta(V + E)$ time, as we shall show in Section 3.8.

In practice, to produce a better quality coloring tends to cost more in running time. That is, the six heuristics, which are listed in increasing order of coloring quality, are also listed in increasing order of running time. The only exception is GREEDY-ID, which is dominated

[2]If the rule for an ordering heuristic allows for ties in the priority function (the priority function is not injective), we shall assume that ties are broken randomly. Formally, suppose that an ordering heuristic H produces a priority function ρ_H which may contain ties. We extend ρ_H to a priority function ρ that maps each vertex $v \in V$ to an ordered pair $\langle \rho_H(v), \rho_{\mathrm{R}}(v) \rangle$, where the priority function ρ_{R} is produced by the random ordering heuristic R. To determine which of two vertices $u, v \in V$ has higher priority, we compare the ordered pairs $\rho(u)$ and $\rho(v)$ lexicographically. Notwithstanding this subtlety, we shall still adopt the simplifying convenience of viewing the priority function as mapping vertices to real numbers. In fact, the range of the priority function can be any linearly ordered set.

```
GREEDY($G$)
let $G = (V, E, \rho)$
for $v \in V$ in order of decreasing $\rho(v)$
    $C \leftarrow \{1, 2, \ldots, \deg(v) + 1\}$
    for $u \in \text{Adj}[v]$ such that $\rho(u) > \rho(v)$
        $C \leftarrow C - \{\text{color}(u)\}$
    $\text{color}(v) \leftarrow \min C$
```

Figure 3-1: Pseudocode for a serial greedy graph-coloring algorithm. Given a vertex-weighted graph $G = (V, E, \rho)$, where the priority of a vertex $v \in V$ is given by $\rho(v)$, GREEDY colors each vertex $v \in V$ in decreasing order according to $\rho(v)$.

```
JP($G$)
let $G = (V, E, \rho)$
parallel for $v \in V$
    $v.pred = \{u \in V : (u, v) \in E \text{ and } \rho(u) > \rho(v)\}$
    $v.succ = \{u \in V : (u, v) \in E \text{ and } \rho(u) < \rho(v)\}$
    $v.counter \leftarrow |v.pred|$
parallel for $v \in V$
    if $v.pred == \emptyset$
        JP-COLOR($v$)
```

```
JP-COLOR($v$)
$\text{color}(v) \leftarrow$ GET-COLOR($v$)
parallel for $u \in v.succ$
    if JOIN($u.counter$) $== 0$
        JP-COLOR($u$)
```

```
GET-COLOR($v$)
$C \leftarrow \{1, 2, \ldots, |v.pred| + 1\}$
parallel for $u \in v.pred$
    $C = C - \{\text{color}(u)\}$
return $\min C$
```

Figure 3-2: The Jones-Plassmann parallel coloring algorithm. JP uses a recursive helper function JP-COLOR to process a vertex once all of its predecessors have been colored. JP-COLOR uses the helper routine GET-COLOR to find the smallest color available to color a vertex v.

by GREEDY-SL in both coloring quality and runtime. The experiments discussed in the Appendix (Section 3.11) summarize our empirical findings for serial greedy coloring.

Parallel greedy coloring

There is a historical tension between coloring quality and the parallel scalability of greedy graph coloring. While the traditional ordering heuristics FF, LF, ID, and SL are efficient using GREEDY, it can be shown that any parallelization of them requires worst-case span of $\Omega(V)$ for a general graph $G = (V, E)$. Of the various attempts to parallelize greedy coloring [236, 96, 74], the algorithm first proposed by Jones and Plassmann [179] extends the greedy algorithm in a straightforward manner, uses work linear in size of the graph, and is deterministic given a random seed. Jones and Plassmann's original paper demonstrates good parallel performance for $O(1)$-degree graphs using the random ordering heuristic R.

Unfortunately, in practice, R tends to produce colorings of relatively poor quality relative to the other traditional ordering heuristics. But the other traditional ordering heuristics are all vulnerable to adversarial graph inputs which cause JP to operate in $\Omega(V)$ time and thus exhibit poor parallel scalability. Consequently, there is need for new ordering heuristics for JP that can achieve both good coloring quality and guaranteed fast parallel performance.

Figure 3-2 gives the pseudocode for JP, which colors a given graph $G = (V, E, \rho)$ in the order specified by the priority function ρ. The algorithm begins in lines 9 and 10 by partitioning the neighbors of each vertex into ***predecessors*** — vertices with larger priorities — and ***successors*** — vertices with smaller priorities. JP uses the recursive JP-COLOR helper function to color a vertex $v \in V$ once all vertices in $v.pred$ have been colored. Initially, lines 12–14 in JP scan the vertices of V to find every vertex that has no predecessors and colors each one using JP-COLOR. Within a call to JP-COLOR(v), line 15 calls GET-COLOR to assign a color to v, and the loop on lines 16–18 broadcasts in parallel to all of v's successors the fact that v is colored. For each successor $u \in v.succ$, line 17 tests whether all of u's predecessors have already been colored, and if so, line 18 recursively calls JP-COLOR on u.

Jones and Plassmann analyze the performance of JP-R for $O(1)$-degree graphs. Although they do not discuss using the naive FF ordering heuristic, it is apparent that there exist adversarial input orderings for which their algorithm would fail to scale. For example, if the graph $G = (V, E)$ is simply a chain of vertices and the input order of V corresponds to their in order in the chain, JP-FF exhibits no parallelism. Jones and Plassmann show that a random ordering produced by R, however, allows the algorithm to run in $O(\lg V / \lg \lg V)$ expected time on this chain graph — and on any $O(1)$-degree graph, for that matter. Section 3.3 of this chapter extends their analysis of JP-R to arbitrary-degree graphs.

Although JP-R scales well in theory, as well as in practice, when it comes to coloring quality, R is one of the weaker ordering heuristics, as we have noted. Of the other heuristics, JP-LF and JP-SL suffer from the same problem as FF, namely, it is possible to construct adversarial graphs that cause them to scale poorly, which we explore in Section 3.4. The ID heuristic tends to produce worse colorings than SL, and since GREEDY-ID also runs more slowly than GREEDY-SL, we have dropped ID from consideration. Moreover, because of our motivation to use the coloring algorithm for online chromatic scheduling, where the performance of the coloring algorithm cannot be sacrificed for marginal improvements in the quality of coloring, we also have dropped the SD heuristic. Since SD produces the best-quality colorings of the six ordering heuristics, however, we see parallelizing it as an interesting opportunity for future research.

Consequently, this chapter focuses on alternatives to the LF and SL ordering heuristics that provide comparable coloring quality while exhibiting the same resilience to adversarial graphs that R shows compared with FF. Specifically, we introduce two new randomized ordering heuristics — "largest log-degree first" (LLF) and "smallest log-degree last" (SLL) — which resemble LF and SL, respectively, but which scale provably well when used with JP. We demonstrate that JP-LLF and JP-SLL provide good parallel scalability in theory and practice and are resilient to adversarial graphs.

Table 3.1 summarizes our empirical findings. The data suggest that the LLF and SLL ordering heuristics produce colorings that are nearly as good as LF and SL, respectively. With respect to performance, our implementations of JP-LLF and JP-SLL actually operate slightly faster on 1 processor than our highly tuned implementations of GREEDY-LF and GREEDY-SL, respectively, and they scale comparably to JP-R.

H	H'	$\frac{C_{H'}}{C_H}$	$\frac{\text{GREEDY-}H}{\text{JP-}H'_1}$	$\frac{\text{JP-}H'_1}{\text{JP-}H'_{12}}$
FF	R	1.011	0.417	7.039
LF	LLF	1.021	1.058	7.980
SL	SLL	1.037	1.092	6.082

Table 3.1: Summary of ordering-heuristic behavior on a suite of 8 real-world graphs and 10 synthetic graphs when run on a machine with 12 Intel Xeon X5650 processor cores. Column H lists three serial heuristics traditionally used for GREEDY, and column H' lists parallel heuristics for JP, of which LLF and SLL are introduced in this chapter. Column "$C_{H'}/C_H$" shows the geometric mean of the ratio of the number of colors the parallel heuristic uses compared to the serial heuristic. Column "GREEDY-H/JP-H'_1" shows the geometric mean of the ratio of serial running times of GREEDY with the serial heuristic versus JP with the analogous parallel heuristic when run on 1 processor. Column "JP-H'_1/JP-H'_{12}" shows the geometric mean of the speedup of each parallel heuristic going from 1 processor to 12.

Outline

The remainder of this chapter is organized as follows. Section 3.2 reviews the asynchronous parallel greedy coloring algorithm first proposed by Jones and Plassmann [179]. We show how JP can be extended to handle arbitrary-degree graphs and arbitrary priority functions. Using work-span analysis [77, Ch. 27], we show that JP colors a Δ-degree graph $G = (V, E, \rho)$ in $\Theta(V + E)$ work and $O(L \lg \Delta + \lg V)$ span, where L is the length of the longest path in G along which the priority function ρ decreases. Section 3.3 analyzes the performance of JP-R, showing that it operates using linear work and $O(\lg V + \lg \Delta \cdot \min\{\sqrt{E}, \Delta + \lg \Delta \lg V/\lg \lg V\})$ span. Section 3.4 shows that there exist "adversarial" graphs for which JP-LF and JP-SL exhibit limited parallel speedup. Section 3.5 analyzes the LLF and SLL ordering heuristics. We show that, given a Δ-degree graph G, JP-LLF colors $G = (V, E, \rho)$ using $\Theta(V + E)$ work and $O(\lg V + \lg \Delta(\min\{\Delta, \sqrt{E}\} + \lg^2 \Delta \lg V/\lg \lg V))$ expected span, while JP-SLL colors $G = (V, E, \rho)$ using same work and an additive $\Theta(\lg \Delta \lg V)$ additional span. Section 3.6 evaluates the performance of JP-LLF and JP-SLL on a suite of 8 real-world and 10 synthetic benchmark graphs. Section 3.7 discusses the software engineering techniques used in our implementation of JP-R, JP-LLF, and JP-SLL. Section 3.8 introduces an algorithm for computing the SD ordering heuristic using $\Theta(V + E)$ work. Section 3.9 discusses related work, and Section 3.10 offers some concluding remarks. The Appendix (Section 3.11) presents some experimental results for serial ordering heuristics.

3.2 The Jones-Plassmann algorithm

This section reviews JP, the parallel greedy coloring algorithm introduced by Jones and Plassmann [179], whose pseudocode is given in Figure 3-2. We first review the dag model of dynamic multithreading and work-span analysis [77, Ch. 27]. Then we describe how JP can be modified from Jones and Plassmann's original algorithm to handle arbitrary-degree graphs and arbitrary priority functions. We analyze JP with an arbitrary priority function ρ and show that on a Δ-degree graph $G = (V, E, \rho)$, JP runs in $\Theta(V + E)$ work and $O(L \lg \Delta + \lg V)$ span, where L is the longest path in the "priority dag" of G induced by ρ.

The dag model of dynamic multithreading

We shall analyze the parallel performance of JP using the dag model of dynamic multithreading introduced by Blumofe and Leiserson [40, 41] and described in tutorial fashion in [77, Ch. 27]. The dag model views the executed computation resulting from running a parallel algorithm as a ***computation dag*** A, in which each vertex denotes an instruction, and edges denote parallel control dependencies between instructions. Although the model encompasses other parallel control constructs, for our purposes, we need only understand that the execution of a **parallel for** loop can be modeled as a balanced binary tree of vertices in the dag, where the leaves of the tree denote the initial instructions of the loop iterations.

To analyze the performance of a dynamic multithreading program theoretically, we assume that the program executes on an ***ideal parallel computer***: each instruction executes in unit time, the computer has ample memory bandwidth, and the computer supports concurrent writes and read-modify-write instructions [151] without incurring overheads due to contention.

Given a dynamic multithreading program whose execution is modeled as a dag A, we can bound the parallel running time $T_P(A)$ of the computation as follows. The ***work*** $T_1(A)$ is the number of strands in the computation dag A. The ***span*** $T_\infty(A)$ is the length of the longest path in A. A deterministic algorithm with work T_1 and span T_∞ can always be executed on P processors in time T_P satisfying $\max\{T_1/P, T_\infty\} \leq T_p \leq T_1/P + T_\infty$ [49, 99, 40, 41, 137]. The ***speedup*** of an algorithm on P processors is T_1/T_P, which is at most P in theory, since $T_P \geq T_\infty$. The ***parallelism*** T_1/T_∞ is the greatest theoretical speedup possible for any number P of processors.

Analysis of JP

To analyze the performance of JP, it is convenient to think of the algorithm as coloring the vertices in the partial order of a "priority dag," similar to the priority dag described by Blelloch *et al.* [34]. Specifically, on a vertex-weighted graph $G = (V, E, \rho)$, the priority function ρ induces a ***priority dag*** $G_\rho = (V, E_\rho)$, where $E_\rho = \{(u, v) \in V \times V : (u, v) \in E \text{ and } \rho(u) > \rho(v)\}$. Notice that G_ρ is a dag, because ρ is an injective function and thus induces a total order on the vertices V. We shall bound the span of JP running on a graph G in terms of the ***depth*** of G_ρ, that is, the length of the longest path through G_ρ. We analyze JP in two steps.

First, we bound the work and span of calls during the execution of JP to the helper routine GET-COLOR(v), which returns the minimum color not assigned to any vertex $u \in v.pred$.

Lemma 12 *The helper routine* GET-COLOR*, shown in Figure 3-2, can be implemented so that during the execution of* JP *on a graph* $G = (V, E, \rho)$*, a call to* GET-COLOR(v) *for a vertex* $v \in V$ *costs* $\Theta(k)$ *work and* $\Theta(\lg k)$ *span, where* $k = |v.pred|$.

PROOF. Implement the set C in GET-COLOR as an array whose ith entry initially stores the value i. The ith element from this array can be removed by setting the ith element to ∞. With this implementation, lines 20–21 execute in $\Theta(k)$ work and $\Theta(\lg k)$ span. The min operation on line 22 can be implemented as a parallel minimum reduction in the same bounds. □

Second, we show that JP colors a graph $G = (V, E, \rho)$ using work $\Theta(V + E)$ and span linear in the depth of the priority dag G_ρ.

Theorem 13 *Given a Δ-degree graph $G = (V, E, \rho)$ for some priority function ρ, let G_ρ be the priority dag induced on G by ρ, and let L be the depth of G_ρ. Then* $\mathrm{JP}(G)$ *runs in* $\Theta(V + E)$ *work and* $O(L \lg \Delta + \lg V)$ *span.*

PROOF. Let us first bound the work and span of JP-COLOR excluding any recursive calls. For a single call to JP-COLOR on a vertex $v \in V$, Lemma 12 shows that line 15 takes $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span. The JOIN operation on line 17 can be implemented as an atomic decrement-and-fetch operation [151] on the specified counter. Hence, excluding the recursive call, the loop on lines 16–18 performs $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span to decrement the counters of all successors of v.

Because JP-COLOR is called once per vertex, the total work that JP spends in calls to JP-COLOR is $\Theta(V + E)$. Furthermore, the span of JP-COLOR is the length of any path of vertices in G_ρ, which is at most L, times $\Theta(\lg \Delta)$. Finally, the loop on lines 8–11 executes in $\Theta(V + E)$ work and $\Theta(\lg V + \lg \Delta)$ span, and the parallel loop on lines 12–14, excluding the call to JP-COLOR, executes in $\Theta(V + E)$ work and $\Theta(\lg V)$ span. □

3.3 JP with random ordering

This section bounds the depth of a priority dag G_ρ induced on a Δ-degree graph $G = (V, E, \rho)$ by a random priority function ρ in R. We show that the expected depth of G_ρ is $O(\min\{\sqrt{E}, \Delta + \lg \Delta \lg V / \lg \lg V\})$. Combined with Theorem 13, this bound implies that the expected span of JP-R is $O(\lg V + \lg \Delta \cdot \min\{\sqrt{E}, \Delta + \lg \Delta \lg V / \lg \lg V\})$. This bound extends Jones and Plassmann's $O(\lg V / \lg \lg V)$ bound for the depth of G_ρ when $\Delta = \Theta(1)$ [179].

To bound the depth of a priority dag G_ρ induced on a graph G by $\rho \in \mathrm{R}$, let us start by bounding the number of length-k paths in G_ρ. Each path in G_ρ corresponds to a unique ***simple*** path in G, that is, a path in which each vertex in G appears at most once. The following lemma bounds the number of length-k simple paths in G.

Lemma 14 *The number of length-k simple paths in any Δ-degree graph $G = (V, E)$ is at most* $|V| \cdot \min\{\Delta^{k-1}, (2|E|/(k-1))^{k-1}\}$.

PROOF. Consider selecting a length-k simple path $p = \langle v_1, \ldots, v_k \rangle$ in G. There are $|V|$ choices for v_1, and for all $i \in \{1, \ldots, k-1\}$, given a choice of $\langle v_1, \ldots, v_i \rangle$, there are at most $\deg(v_i)$ choices for v_{i+1}. Hence there are at most $J = |V| \cdot \prod_{i=1}^{k-1} \deg(v_i)$ simple paths in G of length k. Let $V_k \subseteq V$ denote some set of $k-1$ vertices in V, and let $\delta = \max_{V_{k-1}}\{\sum_{v \in V_{k-1}} \deg(v)/(k-1)\}$ be the maximum average degree of any such set. Then we have $J \le |V| \cdot \delta^{k-1}$.

The proof follows from two upper bounds on δ. First, because $\deg(v) \le \Delta$ for all $v \in V$, we have $\delta \le \Delta$. Second, for all $V_{k-1} \subseteq V$, we have $\sum_{v \in V_{k-1}} \deg(v) \le \sum_{v \in V} \deg(v) = 2|E|$ by the handshaking lemma [77, p. 1172–3], and thus $\delta \le 2|E|/(k-1)$. □

Intuitively, the bound on the expected depth of G_ρ follows by arguing that although the number of simple length-k paths in a graph G might be exponential in k, for sufficiently large k, the probability is tiny that any such path is a path in G_ρ. To formalize this argument, we make use of the following technical lemma.

Lemma 15 *Define the function $g(\alpha, \beta)$ for $\alpha, \beta > 1$ as*

$$g(\alpha, \beta) = e^2 \frac{\ln \alpha}{\ln \beta} \ln\left(e \frac{\beta \ln \alpha}{\alpha \ln \beta}\right) .$$

Then for all $\beta \geq e^2$, $\alpha \geq 2$, and $\beta \geq \alpha$, we have $g(\alpha, \beta) \geq 1$.

PROOF. We consider the cases when $\alpha \geq e^2$ and when $\alpha < e^2$ separately.

When $\alpha > e^2$, the partial derivative of $g(\alpha, \beta)$ with respect to β is

$$\begin{aligned} \frac{\partial g(\alpha, \beta)}{\partial \beta} &= e^2 \frac{\ln \alpha}{\beta \ln^2 \beta} \ln\left(\frac{\alpha}{e^2} \frac{\ln \beta}{\ln \alpha}\right) \\ &\geq 0 , \end{aligned}$$

since $\alpha \ln \beta / e^2 \ln \alpha \geq 1$ when $\alpha \geq e^2$ and $\beta \geq \alpha$. Thus, $g(\alpha, \beta)$ is a nondecreasing function in its second argument when $\alpha \geq e^2$ and $\beta \geq \alpha$. Since we have

$$\begin{aligned} g(\alpha, \alpha) &= e^2 (\ln \alpha / \ln \alpha) \ln(e(\alpha \ln \alpha)/(\alpha \ln \alpha)) \\ &\geq 1 , \end{aligned}$$

it follows that $g(\alpha, \beta) \geq 1$ for $\alpha \geq e^2$ and $\beta \geq \alpha$.

When $e^2 > \alpha \geq 2$, we make use of the fact that $2\beta / e \ln \beta > \sqrt{\beta}$ for all $\beta > e^2$:

$$\begin{aligned} g(\alpha, \beta) &\geq (e^2 \ln 2 / \ln \beta) \ln(2\beta/(e \ln \beta)) \\ &\geq (e^2 \ln 2 / \ln \beta) \ln\left(\sqrt{\beta}\right) \\ &\geq (e^2 \ln 2 \ln \beta)/(2 \ln \beta) \\ &\geq 1 . \end{aligned}$$

□

The following theorem applies Lemmas 14 and 15 to establish the bound on the depth of G_ρ.

Theorem 16 *Let $G = (V, E)$ be a Δ-degree graph, let $n = |V|$ and $m = |E|$, and let G_ρ be a priority dag induced on G by a random priority function $\rho \in \mathrm{R}$. For any constant $\epsilon > 0$ and sufficiently large n, with probability at most $n^{-\epsilon}$, there exists a directed path of length $e^2 \cdot \min\{\Delta, \sqrt{m}\} + (1 + \epsilon) \min\{e^2 \ln \Delta \ln n / \ln \ln n, \ln n\}$ in G_ρ.*

PROOF. Let $p = \langle v_1, \ldots, v_k \rangle$ be a length-k simple path in G. Because ρ is a random priority function, ρ induces each possible permutation among $\{v_1, \ldots, v_k\}$ with equal probability. If p is a directed path in G_ρ, then we must have that $\rho(v_1) < \rho(v_2) < \cdots < \rho(v_k)$. Hence, p is a length-k path in G_ρ with probability at most $1/k!$. If J is the number of length-k simple paths in G, then by the union bound, the probability that a length-k directed path exists in G_ρ is at most $J/k!$, which is at most $J(e/k)^k$ by Stirling's approximation [77, p. 57].

We consider cases when $\Delta < \ln n$ and $\Delta \geq \ln n$ separately. First, suppose that $\Delta < \ln n$. By Lemma 14, the number of length-k simple paths in G is at most $n\Delta^{k-1} \leq n\Delta^k$. By the union bound, the probability that a length-k path exists in G_ρ is at most $n(e\Delta/k)^k$. We assume, without loss of generality, that $\Delta > 2$, since the theorem holds for $O(1)$-degree graphs as a result of [179].

For $\Delta \geq 2$, observe that, by Lemma 15, the function $g(\alpha, \beta) = e^2(\ln \alpha / \ln \beta) \ln(\beta \ln \alpha / \alpha \ln \beta)$ is at least 1 for all $\alpha \geq 2$ and $\beta \geq e^2$. Letting $\alpha = \Delta$, $\beta = \ln n$, and $k = e^2(\Delta + (1 + \epsilon) \ln \Delta \ln n / \ln \ln n)$, we conclude that

$$\begin{aligned} n(e\Delta/k)^k &= n \cdot \exp(-k \ln(k/e\Delta)) \\ &\leq n \cdot \exp\left(-e^2(1+\epsilon) \ln n \frac{\ln \Delta}{\ln \ln n} \ln\left(e \frac{\ln n \ln \Delta}{\Delta \ln \ln n}\right)\right) \\ &= n \cdot \exp(-(1+\epsilon)(\ln n) \cdot g(\Delta, \ln n)) \\ &\leq n e^{-(1+\epsilon)\ln n} \\ &= n^{-\epsilon} \ . \end{aligned}$$

Next, given $\Delta \geq \ln n$, consider the cases when $\Delta < \sqrt{m}$ and $\Delta \geq \sqrt{m}$, separately. When $\Delta < \sqrt{m}$, letting $k = e^2\Delta + (1+\epsilon)\ln n$, the theorem follows from the facts that $k \geq (1+\epsilon)\ln n$ and $k \geq e^2\Delta$. When $\Delta \geq \sqrt{m}$, let $k = e^2\sqrt{m} + (1+\epsilon)\ln n$. By Lemma 14, the number of length-k simple paths is at most $n(2m/(k-1))^{k-1} \leq n(4m/k)^k$, and thus the probability that a length-k path exists in G_ρ is at most $n(4em/k^2)^k$. The theorem follows from the facts that $k \geq (1+\epsilon)\ln n$ and $k^2 \geq e^4 m$. □

Corollary 17 *Given a graph $G = (V, E, \rho)$, where $\rho \in$ R is a random priority function, the expected depth of the priority dag G_ρ is $O(\min\{\sqrt{E}, \Delta + \lg \Delta \lg V / \lg \lg V\})$, and thus* JP-R *colors all vertices of G with $O(\lg V + \lg \Delta \cdot \min\{\sqrt{E}, \Delta + \lg \Delta \lg V / \lg \lg V\})$ expected span.*

PROOF. Theorems 13 and 16 imply the corollary. □

3.4 The LF and SL heuristics

This section shows that the largest-first (LF) and smallest-last (SL) ordering heuristics can inhibit parallel speedup when used by JP. We examine a "clique-chain" graph and show that JP-LF incurs $\Omega(\Delta^2)$ span to color a Δ-degree clique-chain graph $G = (V, E)$, whereas JP-R colors G incurring only $O(\Delta \lg \Delta + \lg^2 \Delta \lg V / \lg \lg V)$ expected span. We formally review the SL ordering heuristic and observe that this formulation of SL means that JP-SL requires $\Omega(V)$ span to color a path graph $G = (V, E)$.

Tables 3.2 and 3.3 summarize the performance of FF, LF and SL on a suite of 8 real-world and 10 synthetic benchmark graphs. The number of edges, ratio of edges to vertices and maximum degree of each benchmark graph is given in Table 3.5.

The LF ordering heuristic

The LF ordering heuristic colors the vertices of a graph $G = (V, E, \rho)$ for some ρ in LF in order of decreasing degree. Formally, $\rho \in$ LF is defined for a vertex $v \in V$ as $\rho(v) = \langle \deg(V), \rho_{\mathrm{R}}(v) \rangle$, where ρ_{R} is randomly chosen from R.

Although LF has been used in parallel greedy graph-coloring algorithms in the past [4, 129], Figure 3-3 illustrates a Δ-degree "clique-chain" graph $G = (V, E)$ for which JP-LF incurs $\Omega(\Delta^2)$ span to color, but JP-R colors with only $O(\Delta \lg \Delta + \lg^2 \Delta \lg V / \lg \lg V)$ expected span. Conceptually, the ***clique-chain*** graph comprises a set of cliques of increasing size that are connected in a "chain" such that JP-LF is forced to color these cliques sequentially from largest to smallest. Figure 3-3 illustrates a Δ-degree clique-chain graph $G = (V, E)$,

			Greedy	JP			
Graph	H	C_H	T_S	T_1	T_{12}	T_S/T_1	T_1/T_{12}
com-orkut	FF	175	2.23	4.16	0.817	0.54	5.09
	LF	87	3.54	6.43	1.067	0.55	6.02
	SL	83	10.59	12.94	8.264	0.82	1.57
liveJournal1	FF	352	0.89	1.69	0.275	0.52	6.15
	LF	323	2.34	2.89	0.365	0.81	7.91
	SL	322	4.69	4.76	2.799	0.98	1.70
europe-osm	FF	5	1.32	∞	∞	∞	∞
	LF	4	17.15	5.16	0.587	3.33	8.79
	SL	3	19.87	∞	∞	∞	∞
cit-Patents	FF	17	0.50	0.99	0.152	0.50	6.47
	LF	14	2.00	1.52	0.211	1.31	7.22
	SL	13	3.21	3.05	1.579	1.05	1.93
as-skitter	FF	103	0.24	0.55	0.109	0.45	5.00
	LF	71	2.43	0.69	0.133	3.51	5.21
	SL	70	2.79	1.19	0.733	2.35	1.62
wiki-Talk	FF	102	0.09	0.23	0.046	0.38	4.99
	LF	72	0.49	0.37	0.073	1.30	5.12
	SL	56	0.61	0.57	0.293	1.08	1.93
web-Google	FF	44	0.09	0.20	0.036	0.47	5.62
	LF	45	0.25	0.29	0.042	0.88	6.85
	SL	44	0.47	0.53	0.278	0.89	1.92
com-youtube	FF	57	0.06	0.16	0.027	0.39	6.07
	LF	32	0.25	0.24	0.040	1.03	6.12
	SL	28	0.35	0.36	0.181	0.98	1.99

Table 3.2: Performance measurements for a set of real-world graphs taken from Stanford's SNAP project [220]. The column heading H denotes that the priority function used for the experiment in a particular row was produced by the ordering heuristic listed in the column. The average number of colors used by the corresponding ordering heuristic and graph is C_H. The time in seconds of Greedy, JP with 1 worker and with 12 workers is given by T_S, T_1 and T_{12}, respectively, where a value of ∞ indicates that the program crashed due to excessive stack usage. Details of the experimental setup and graph suite can be found in Section 3.6.

where 3 evenly divides Δ. This clique-chain graph contains a sequence of cliques $\mathcal{K} = \{K_1, K_4, \ldots, K_{\Delta-2}\}$ of increasing size, each pair of which is separated by two additional vertices forming a linear chain. Specifically, for $r \in \{1, 4, \ldots, \Delta - 2\}$, each vertex $u \in K_r$ is connected to each vertex $u \in K_{r+3}$ by a path $\langle u, x_{r+1}, x_{r+2}, v\rangle$ for distinct vertices $x_{r+1}, x_{r+2} \in V$. Additional vertices, shown above the chain in Figure 3-3, ensure that the degree of each vertex in K_r is $r + 2$, and the degrees of the vertices x_{r+1} and x_{r+2} are $r + 3$ and $r + 4$, respectively. Clique-chain graphs of other degrees are structured similarly.

Theorem 18 *For any $\Delta > 0$, there exists a Δ-degree graph $G = (V, E)$ such that* JP-LF

			Greedy	JP			
Graph	H	C_H	T_S	T_1	T_{12}	T_S/T_1	T_1/T_{12}
constant1M	FF	33	0.90	1.70	0.230	0.53	7.40
	LF	32	1.16	2.96	0.386	0.39	7.68
	SL	34	2.96	5.09	2.023	0.58	2.52
constant500K	FF	52	0.74	1.26	0.286	0.59	4.42
	LF	52	0.84	2.55	0.444	0.33	5.73
	SL	53	1.97	3.50	1.435	0.56	2.44
graph500-5M	FF	220	1.83	2.86	0.560	0.64	5.11
	LF	159	3.69	3.99	0.649	0.92	6.15
	SL	158	8.43	9.45	5.576	0.89	1.69
graph500-2M	FF	206	0.52	0.98	0.208	0.53	4.72
	LF	153	0.98	1.34	0.221	0.73	6.06
	SL	153	2.22	2.72	1.559	0.81	1.75
rMat-ER-2M	FF	12	0.47	1.11	0.169	0.42	6.60
	LF	11	1.07	1.72	0.204	0.62	8.45
	SL	11	2.22	3.07	1.362	0.72	2.25
rMat-G-2M	FF	27	0.48	0.88	0.130	0.55	6.74
	LF	15	1.18	1.42	0.200	0.83	7.09
	SL	15	2.59	3.09	1.712	0.84	1.81
rMat-B-2M	FF	105	0.50	0.84	0.151	0.60	5.53
	LF	67	1.00	1.28	0.191	0.79	6.68
	SL	67	2.41	2.84	1.691	0.85	1.68
big3dgrid	FF	4	0.41	1.68	0.173	0.24	9.69
	LF	7	4.07	1.53	0.198	2.66	7.72
	SL	7	4.77	2.60	1.074	1.83	2.42
cliqueChain400	FF	399	0.05	0.09	0.224	0.51	0.40
	LF	399	0.05	∞	∞	∞	∞
	SL	399	0.08	0.14	0.265	0.55	0.54
path-10M	FF	2	0.18	∞	∞	∞	∞
	LF	3	2.49	0.76	0.092	3.26	8.27
	SL	2	2.58	∞	∞	∞	∞

Table 3.3: Performance measurements for five classes of synthetically generated graphs: constant degree, rMat, 3D grid, clique chain and path. The column headings are equivalent to those in Table 3.2.

colors G in $\Omega(\Delta^2)$ span and JP-R *colors G in $O(\Delta \lg \Delta + \lg^2 \Delta \lg V / \lg \lg V)$ expected span.*

Proof. Assume without loss of generality that 3 evenly divides Δ and that G is a clique-chain graph. The span of JP-R follows from Corollary 3.3. Because JP-LF trivially requires $\Omega(1)$ span to process each vertex in G, the span of JP-LF on G can be bounded by showing that the length of the longest path p in the priority dag G_ρ induced on G by any priority function ρ in LF is $\Delta^2/6 + \Delta/2 + 2$. Because LF assigns higher priority to higher-degree

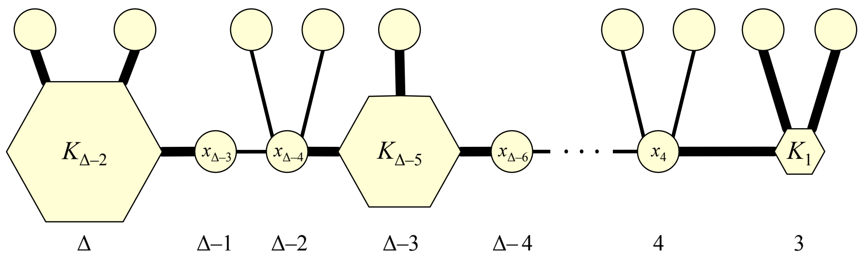


Figure 3-3: A Δ-degree clique-chain graph G, which Theorem 18 shows is adversarial for JP-LF. This graph contains $\Theta(\Delta^2)$ vertices arranged as a chain of cliques. Each hexagon labeled K_r represents a clique of r vertices, and circles represent individual vertices. A thick edge between an individual vertex and a clique indicates that the vertex is connected to every vertex within the clique. A label below an individual vertex indicates the degree of the associated vertex, and a label below a clique indicates the degree of every vertex within that clique.

vertices, p starts at some vertex in $K_{\Delta-2}$, which has degree Δ, and passes through the $\Delta-2$ vertices in $K_{\Delta-2}$ followed by $x_{\Delta-3}$ and $x_{\Delta-4}$.[3] The remainder of p is a longest path through the clique-chain graph G' of degree $\Delta-3$ in the remaining graph $G - K_{\Delta-2} - \{x_{\Delta-3}, x_{\Delta-4}\}$, which has a longest path p' of length $|p'| = (\Delta-3)^2/6 + (\Delta-3)/2 + 2$ by induction. The length of p is thus $\Delta + |p'| = \Delta^2/6 + \Delta/2 + 2$. □

The SL ordering heuristic

We focus on the formulation of the SL ordering heuristic due to Allwright *et al.* [4], because our experiments indicate that it gives colorings using fewer colors than other formulations [246].

Given a graph $G = (V, E)$, the SL ordering heuristic produces a priority function ρ via an iterative algorithm that assigns priorities to the vertices V in rounds to induce an ordering on V. For $i \geq 0$, let $G_i = (V_i, E_i)$ denote the subgraph of G remaining at the start of round i, and let δ_i denote an upper bound on the smallest degree of any vertex $v \in V_i$. Assume that $\delta_0 = 1$. At the start of round i, remove all vertices $v \in V_i$ such that $\deg(v) \leq \max\{\delta_{i-1}, \min_{v \in V_i}\{\deg(v)\}\}$. For a vertex v removed in round i, a priority function $\rho \in \mathrm{SL}$ is defined as $\rho(v) = \langle i, \rho_{\mathrm{R}}(v)\rangle$ where $\rho_{\mathrm{R}} \in \mathrm{R}$ is a random priority function.

The following theorem shows that there exist graphs for which JP-SL incurs a large span, whereas JP-R incurs only a small span.

Theorem 19 *There exists a class of graphs such that for any $G = (V, E, \rho)$ in the class and for any priority function $\rho \in \mathrm{SL}$,* JP-SL *incurs $\Omega(V)$ span and* JP-R *incurs $O(\lg V / \lg\lg V)$ span.*

PROOF. Consider the algorithm to compute the priority function ρ for all vertices in a path graph G. By induction over the rounds, the graph G_i at the start of round i is a path with $|V| - 2i + 2$ vertices, and in round i the 2 vertices at the endpoints of G_i will be removed. Hence $\lceil |V|/2 \rceil$ rounds are required to assign priorities for all vertices in G. A

[3] Notice that it does not matter how ties are broken in the priority function.

similar argument shows that the resulting priority dag G_ρ contains a path of length $|V|/2$ along which the priorities strictly decrease. JP-SL trivially incurs $\Omega(1)$ span through each vertex in the longest path in G_ρ. Since there are $\Theta(V)$ total vertices along the path and by Corollary 3.3 with $\Delta = \Theta(1)$, the theorem follows. □

We shall see in Section 3.5 that it is possible to achieve coloring quality comparable to LF and SL, but with guaranteed parallel performance comparable to JP-R.

3.5 Log ordering heuristics

This section describes the largest-log-degree-first (LLF) and smallest-log-degree-last (SLL) ordering heuristics. Given a Δ-degree graph G, we show that the expected depth of the priority dag G_ρ induced on G by a priority function $\rho \in$ LLF is $O(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta \lg V/\lg\lg V)$. The same bound applies to the depth of a priority dag G_ρ induced on a graph G by a priority function $\rho \in$ SLL, though $O(\lg\Delta\lg V)$ additional span is required to calculate ρ using the method given in Figure 3-4. Combined with Theorem 13, these bounds imply that the expected span of JP-LLF is $O(\lg V + \lg\Delta(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta\lg V/\lg\lg V))$ and the expected span of JP-SLL is $O(\lg\Delta\lg V + \lg\Delta(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta\lg V/\lg\lg V))$.

The LLF ordering heuristic

The **LLF *ordering heuristic*** orders the vertices in decreasing order by the logarithm of their degree. More precisely, given a graph $G = (V, E, \rho)$ for some $\rho \in$ LLF, the priority of each $v \in V$ is equal to $\rho(v) = \langle\lceil\lg(\deg(v))\rceil, \rho_{\mathrm{R}}(v)\rangle$, where $\rho_{\mathrm{R}} \in$ R is a random priority function and $\lg x$ denotes $\log_2 x$. [4] For a given graph G, the following theorem bounds the depth of the priority dag G_ρ induced by $\rho \in$ LLF.

Theorem 20 *Let $G = (V, E)$ be a Δ-degree graph, and let G_ρ be the priority dag induced on G by a priority function $\rho \in$ LLF. The expected length of the longest directed path in G_ρ is $O(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta\lg V/\lg\lg V)$.*

PROOF. Consider a length-k path $p = \langle v_1, \ldots, v_k\rangle$ in G_ρ. Let $G(\ell) \subseteq G_\rho$ be the subdag of G_ρ induced by those vertices $v \in V$ for which $\rho(v) = \lceil\lg(\deg(v))\rceil = \ell$. Suppose that $v_i \in G(\ell)$ for some $v_i \in p$. Since $\lceil\lg(\deg(v_{i-1}))\rceil \geq \lceil\lg(\deg(v_i))\rceil$ for all $i > 1$, we have $v_{i-1} \in G(\ell')$ for some $\ell' \geq \ell$. We can therefore decompose p into a sequence of paths $p = \langle p_{\lceil\lg\Delta\rceil}, \ldots, p_0\rangle$ such that each subpath $p_\ell \in p$ is a path through $G(\ell)$. By definition of LLF, the subdag $G(\ell)$ is a dag induced on a graph with degree 2^ℓ by a random priority function.

By Corollary 3.3, the expected length of p_ℓ is $O(2^\ell + \ell\lg V/\lg\lg V)$. Linearity of expec-

[4] The theoretical results in this section assume only that the base b of the logarithm is a constant. In practice, however, it is possible that the choice of b could have impact on the coloring quality or runtime of JP-LLF. We studied this trade-off and found that there is only a minor dependence on b. In general, the coloring quality and runtime of JP-LLF smoothly transitions from the behavior of JP-LF for small b and the behavior of JP-R for large b, sweeping out a Pareto-efficient frontier of reasonable choices. We chose $b = 2$ for our experiments, because $\log_2 x$ can be calculated conveniently by native instructions on modern architectures.

tation therefore implies that

$$\begin{aligned}\mathrm{E}[|p|] &= \sum_{\ell=0}^{\lceil \lg \Delta \rceil} O\left(2^\ell + \ell \lg V / \lg \lg V\right) \\ &= O\left(\Delta + \lg^2 \Delta \lg V / \lg \lg V\right) .\end{aligned}$$

To establish the $\sqrt{E}$ bound, observe that at most $E/2^\ell$ vertices have degree at least 2^ℓ. Consequently, for $\ell > \lg \sqrt{E}$, the depth of $G(\ell)$ can be at most $E/2^\ell$. Hence we have

$$\begin{aligned}\mathrm{E}[|p|] &\leq \sum_{\ell=0}^{\lceil \lg \sqrt{E} \rceil} O\left(2^\ell\right) + \sum_{\ell=\lceil \lg \sqrt{E} \rceil}^{\infty} E/2^\ell + \sum_{\ell=0}^{\lceil \lg \Delta \rceil} O(\ell \lg V / \lg \lg V) \\ &= O\left(\sqrt{E} + \lg^2 \Delta \lg V / \lg \lg V\right).\end{aligned}$$

□

Corollary 21 *Given a graph $G = (V, E, \rho)$ for some $\rho \in$ LLF, JP-LLF colors all vertices in G with expected span $O(\lg V + \lg \Delta(\min\{\sqrt{E}, \Delta\} + \lg^2 \Delta \lg V / \lg \lg V))$.* □

PROOF. The corollary follows from Theorem 13. □

The SLL ordering heuristic

To understand the **SLL *ordering heuristic***, it is convenient to consider in isolation how to compute its priority function. The pseudocode in Figure 3-4 for SLL-ASSIGN-PRIORITIES describes algorithmically how to perform this computation on a given graph $G = (V, E)$. As Figure 3-4 shows, a priority function $\rho \in$ SLL can be computed by iteratively removing low-degree vertices from G in rounds. The priority of a vertex $v \in V$ is the round number in which v is removed, with ties broken randomly. As with SL, SLL colors the vertices of G in the reverse order in which they are removed, but SLL-ASSIGN-PRIORITIES determines when to remove a vertex using a degree bound that grows exponentially. SLL-ASSIGN-PRIORITIES considers each degree bound for a maximum of r rounds. Effectively, a vertex is removed from G based on the logarithm of its degree in the remaining graph.

We can formalize the behavior of SLL as follows. Given a graph G, let $G_i = (V_i, E_i)$ denote the subgraph of G remaining at the start of round i. As Figure 3-4 shows, for each $d \in \{0, 1, \ldots, \lg \Delta\}$, SLL-ASSIGN-PRIORITIES executes r rounds in which it removes vertices $v \in V_i$ such that $\deg(v) \leq 2^d$ in G_i.[5]

For a given graph G, the following theorem bounds the depth of the priority dag G_ρ induced by a priority function $\rho \in$ SLL.

Theorem 22 *Let $G = (V, E)$ be a Δ-degree graph, and let G_ρ be the priority dag induced on G by a random priority function $\rho \in$ SLL. The expected length of the longest directed path in G_ρ is $O(\min\{\Delta, \sqrt{E}\} + \lg^2 \Delta \lg V / \lg \lg V)$.*

[5]As with LLF, the degree cutoff 2^d on line 30 of Figure 3-4 could be b^d for an arbitrary constant base b with no harm to the theoretical results. We explored the choice of base empirically, but found that there was only a minor dependence on b. Generally, JP-SLL smoothly transitions from the behavior of JP-SL for small b to the behavior of JP-R and for large b. We therefore chose $b = 2$ for our experiments because of its implementation simplicity.

```
SLL-ASSIGN-PRIORITIES$(G, r)$
23  let $G = (V, E)$
24  $i \leftarrow 1$
25  $U \leftarrow V$
26  let $\Delta$ be the degree of $G$
27  let $\rho_{\mathrm{R}} \in \mathrm{R}$ be a random priority function
28  for $d \leftarrow 0$ to $\lg \Delta$
29      for $j \leftarrow 1$ to $r$
30          $Q \leftarrow \{u \in U : |\mathrm{Adj}[u] \cap U| \leq 2^d\}$
31          parallel for $v \in Q$
32              $\rho(v) \leftarrow \langle i, \rho_{\mathrm{R}}(v) \rangle$
33          $U \leftarrow U - Q$
34          $i \leftarrow i + 1$
35  return $\rho$
```

Figure 3-4: Pseudocode for SLL-ASSIGN-PRIORITIES, which computes a priority function $\rho \in$ SLL for the input graph. The input parameter r denotes the maximum number of times SLL-ASSIGN-PRIORITIES is permitted to remove vertices of at most a particular degree 2^d on lines 29–34.

PROOF. We begin with an argument similar to the proof of Theorem 20. Let $p = \langle v_1, \ldots, v_k \rangle$ be a length-k path in G_ρ, and let $G(\ell) \subseteq G_\rho$ be the subdag of G_ρ induced by those vertices $v \in V$, where $\rho(v) = \ell$. Since lines 29–34 of SLL-ASSIGN-PRIORITIES remove vertices with degree at most 2^d exactly r times for each $d \in [0, \ldots, \lg \Delta]$, we have that $\lfloor \rho(v)/r \rfloor = d$, and thus the degree of $G(\ell)$ is at most $2^{\lfloor \ell/r \rfloor}$. Suppose that $v_i \in G(\ell)$ for some $v_i \in p$. Since $\rho(v_{i-1}) \leq \rho(v_i)$ for all $i > 1$, we have $v_{i-1} \in G(\ell')$ for some $\ell' \geq \ell$. We can therefore decompose p into a sequence of paths $p = \langle p_{\lceil r \lg \Delta \rceil}, \ldots, p_0 \rangle$ where each $p_\ell \in p$ is a path in $G(\ell)$. By definition of SLL, the subdag $G(\ell)$ is a dag induced on a subgraph with degree at most $2^{\lfloor \ell/r \rfloor}$ by a random priority function.

By Corollary 3.3, the expected length of p_ℓ is $O(2^{\lfloor \ell/r \rfloor} + \lfloor \ell/r \rfloor \lg V / \lg\lg V)$. Linearity of expectation therefore implies that

$$\begin{aligned} \mathrm{E}[|p|] &= \sum_{\ell=0}^{\lceil r \lg \Delta \rceil} O\Big(2^{\lfloor \ell/r \rfloor} + \lfloor \ell/r \rfloor \lg V / \lg\lg V\Big) \\ &= O\big(\Delta + \lg^2\Delta \lg V / \lg\lg V\big) \ . \end{aligned}$$

Next, because at most $E/2^{\lfloor \ell/r \rfloor}$ vertices can have degree at least $2^{\lfloor \ell/r \rfloor}$, we have for $\ell > r \lg \sqrt{E}$ that the longest path through the subdag $G(\ell)$ is no longer than $E/2^{\lfloor \ell/r \rfloor}$. We thus conclude that

$$\begin{aligned}
\mathrm{E}[|p|] &\leq \sum_{\ell=0}^{\lceil r \lg \sqrt{E} \rceil} O\left(2^{\lfloor \ell/r \rfloor}\right) + \sum_{\ell=\lceil r \lg \sqrt{E} \rceil}^{\infty} E/2^{\lfloor \ell/r \rfloor} \\
&\quad + \sum_{\ell=0}^{\lceil r \lg \Delta \rceil} O(\lfloor \ell/r \rfloor \lg V / \lg \lg V) \\
&= O\left(\sqrt{E} + \lg^2 \Delta \lg V / \lg \lg V\right) .
\end{aligned}$$

□

Corollary 23 *Given a graph* $G = (V, E, \rho)$ *for some* $\rho \in \mathrm{SLL}$, JP-SLL *colors all vertices in* G *with expected span* $O(\lg \Delta \lg V + \lg \Delta(\min\{\sqrt{E}, \Delta\} + \lg^2 \Delta \lg V / \lg \lg V))$.

PROOF. The procedure SLL-ASSIGN-PRIORITIES calls the parallel loop on line 31 $O(\lg \Delta)$ times, each of which has expected span $O(\lg V)$. The proof then follows from Theorems 13 and 22. □

3.6 Empirical evaluation

This section evaluates the LLF and SLL ordering heuristics empirically using a suite of eight real-world and ten synthetic graphs. We describe the experimental setup used to evaluate JP-R, JP-LLF, and JP-SLL, and we compare their performance with GREEDY-FF, GREEDY-LF, and GREEDY-SL. We compare the ordering heuristics in terms of the quality of the colorings they produce and their execution times. We conclude that LLF and SLL produce colorings with quality comparable to LF and SL, respectively, and that JP-LLF and JP-SLL scale well. We also show that the engineering quality of our implementations appears to be competitive with COLPACK [121], a publicly available graph-coloring library. Our source code and data are available from `http://supertech.csail.mit.edu`.

Experimental setup

To evaluate the ordering heuristics, we implemented JP using Intel Cilk Plus [171] and engineered it to use the parallel ordering heuristics R, LLF, and SLL. To compare these parallel codes against their serial counterparts, we implemented GREEDY in C to use the FF, LF, or SL ordering heuristics. In order to empirically evaluate the potential parallel performance of the serial ordering heuristics, we also engineered JP to use FF, LF, or SL. We evaluated our implementations on a dual-socket Intel Xeon X5650 with a total of 12 processor cores operating at 2.67-GHz (hyperthreading disabled); 49 GB of DRAM; 2 12-MB L3-caches, each shared between 6 cores; and private L2- and L1-caches with 128 KB and 32 KB, respectively. Each measurement was taken as the median of 7 independent trials, and the averages of those measurements reported in Tables 3.6 and 3.7 were taken across 5 independent random seeds.

These implementations were run on a suite of eight real-world graphs and ten synthetic graphs. The real-world graphs came from the Large Network Dataset Collection provided by Stanford's SNAP project [220]. The synthetic graphs consist of the adversarial graphs

Graph	$\|V\|$	a	b	c	d
graph500-5M	5M	0.57	0.19	0.19	0.05
graph500-2M	2M	0.57	0.19	0.19	0.05
rMat-ER-2M	2M	0.25	0.25	0.25	0.25
rMat-G-2M	2M	0.45	0.15	0.15	0.25
rMat-B-2M	2M	0.55	0.15	0.15	0.15

Table 3.4: Parameters for the generation of rMat graphs [65], where $a + b + c + d = 1$ and $b = c$, when the desired graph is undirected. An rMat graph is built by adding $|E|$ edges independently at random using the following rule: Let k be the number of 1's in a binary representation of i. As each edge is added, the probability that the ith vertex v_i is selected as an endpoint is $(a + c)^k (b + d)^{\lg n - k}$.

Graph	$\|E\|$	$\|E\|/\|V\|$	Δ
com-orkut	117.2M	38.1	33,313
liveJournal1	42.9M	8.8	20,333
europe-osm	36.0M	0.7	9
cit-Patents	16.5M	2.7	793
as-skitter	11.1M	1.0	35,455
wiki-Talk	4.7M	1.9	100,029
web-Google	4.3M	4.7	6,332
com-youtube	3.0M	2.6	28,754
constant1M	50.0M	50.0	100
constant500K	50.0M	99.9	200
graph500-5M	49.1M	5.9	121,495
graph500-2M	19.2M	9.2	70,718
rMat-ER-2M	20.0M	9.5	44
rMat-G-2M	20.0M	9.5	938
rMat-B-2M	19.8M	9.4	14,868
big3dgrid	29.8M	3.0	6
cliqueChain400	3.6M	132.4	400
path-10M	10.0M	1.0	2

Table 3.5: Number of edges, ratio of edges to vertices and maximum vertex degree for a collection of real-world and synthetic graphs, which lie above and below the center line, respectively.

described in Section 3.4 and a set of graphs from three classes: constant degree, 3D grid, and "recursive matrix" (rMat) [65, 62]. The adversarial graphs — cliqueChain400 and path-10M — are described in Figure 3-3 with $\Delta = 400$ and Theorem 19 with $|V| = 10,000,000$, respectively. The constant-degree graphs — constant1M and constant500K — have 1M and 500K vertices and constant degrees of 100 and 200, respectively. These graphs were generated such that every pair of vertices is equally likely to be connected and every vertex has the same degree. The graph big3dgrid is a 3-dimensional grid on 10M vertices. The rMat graphs were generated using the parameters in Table 3.4.

			GREEDY			JP			
Graph	H	C_H	T_S	H'	$C_{H'}$	T_1	T_{12}	T_S/T_1	T_1/T_{12}
com-orkut	FF	175	2.23	R	132	4.44	0.817	0.50	5.43
	LF	87	3.54	LLF	98	5.74	0.846	0.62	6.79
	SL	83	10.59	SLL	84	9.90	1.865	1.07	5.31
liveJournal1	FF	352	0.89	R	330	2.08	0.231	0.43	8.98
	LF	323	2.34	LLF	326	2.23	0.286	1.05	7.80
	SL	322	4.69	SLL	327	4.03	0.704	1.16	5.73
europe-osm	FF	5	1.32	R	5	4.04	0.391	0.33	10.34
	LF	4	17.15	LLF	4	4.93	0.473	3.48	10.41
	SL	3	19.87	SLL	3	7.28	1.232	2.73	5.91
cit-Patents	FF	17	0.50	R	21	1.08	0.163	0.46	6.67
	LF	14	2.00	LLF	14	1.46	0.160	1.37	9.11
	SL	13	3.21	SLL	14	2.90	0.519	1.11	5.58
as-skitter	FF	103	0.24	R	81	0.58	0.114	0.42	5.07
	LF	71	2.43	LLF	72	0.63	0.106	3.84	5.99
	SL	70	2.79	SLL	71	1.04	0.269	2.67	3.88
wiki-Talk	FF	102	0.09	R	85	0.28	0.053	0.31	5.28
	LF	72	0.49	LLF	70	0.34	0.050	1.43	6.78
	SL	56	0.61	SLL	62	0.55	0.124	1.12	4.43
web-Google	FF	44	0.09	R	44	0.21	0.029	0.44	7.44
	LF	45	0.25	LLF	44	0.27	0.030	0.94	8.92
	SL	44	0.47	SLL	44	0.50	0.093	0.94	5.44
com-youtube	FF	57	0.06	R	46	0.18	0.026	0.36	6.86
	LF	32	0.25	LLF	33	0.22	0.028	1.11	7.97
	SL	28	0.35	SLL	28	0.35	0.073	1.01	4.75

Table 3.6: Performance measurements for a set of real-world graphs taken from Stanford's SNAP project [220]. The column heading H denotes that the priority function used for the experiment in a particular row was produced by the ordering heuristic listed in the column. The average number of colors used by the corresponding ordering heuristic and graph is C_H. The time in seconds of GREEDY, JP with 1 worker and with 12 workers is given by T_S, T_1 and T_{12}, respectively. Details of the experimental setup and graph suite can be found in Section 3.6.

Coloring quality of R, LLF, and SLL

Tables 3.6 and 3.7 present the coloring quality of the three parallel ordering heuristics R, LLF, and SLL alongside that of their serial counterparts FF, LF, and SL.

The number of colors used by LLF was comparable to that used by LF on the vast majority of the 18 graphs. Indeed, LLF produced colorings that were within 2 colors of LF on all synthetic graphs and all but 2 real-world graphs: com-orkut and liveJournal1. Similarly, SLL produced colorings that were within 3 colors of SL for all synthetic graphs and all but 2 real-world graphs: liveJournal1 and wiki-Talk.

The liveJournal1 graph appears to benefit little from the ordering heuristics we con-

		Greedy				JP			
Graph	H	C_H	T_S	H'	$C_{H'}$	T_1	T_{12}	T_S/T_1	T_1/T_{12}
constant1M	FF	33	0.90	R	32	1.93	0.255	0.47	7.55
	LF	32	1.16	LLF	32	2.70	0.323	0.43	8.35
	SL	34	2.96	SLL	32	4.63	0.610	0.64	7.59
constant500K	FF	52	0.74	R	52	1.50	0.190	0.49	7.89
	LF	52	0.84	LLF	52	2.01	0.273	0.42	7.34
	SL	53	1.97	SLL	52	3.33	0.498	0.59	6.69
graph500-5M	FF	220	1.83	R	220	2.99	0.558	0.61	5.35
	LF	159	3.69	LLF	160	3.74	0.542	0.99	6.89
	SL	158	8.43	SLL	162	7.63	1.056	1.10	7.23
graph500-2M	FF	206	0.52	R	208	1.01	0.212	0.51	4.77
	LF	153	0.98	LLF	154	1.24	0.151	0.79	8.19
	SL	153	2.22	SLL	156	2.25	0.324	0.99	6.94
rMat-ER-2M	FF	12	0.47	R	12	1.25	0.149	0.37	8.40
	LF	11	1.07	LLF	12	1.63	0.198	0.66	8.25
	SL	11	2.22	SLL	11	3.13	0.506	0.71	6.18
rMat-G-2M	FF	27	0.48	R	27	0.91	0.144	0.53	6.33
	LF	15	1.18	LLF	17	1.34	0.204	0.88	6.54
	SL	15	2.59	SLL	15	2.75	0.432	0.94	6.36
rMat-B-2M	FF	105	0.50	R	105	0.86	0.149	0.58	5.78
	LF	67	1.00	LLF	68	1.18	0.149	0.85	7.94
	SL	67	2.41	SLL	68	2.38	0.376	1.01	6.31
big3dgrid	FF	4	0.41	R	7	1.66	0.178	0.25	9.31
	LF	7	4.07	LLF	7	1.89	0.216	2.15	8.76
	SL	7	4.77	SLL	7	2.63	0.307	1.81	8.57
cliqueChain400	FF	399	0.05	R	399	0.09	0.012	0.50	7.77
	LF	399	0.05	LLF	399	0.12	0.015	0.41	7.70
	SL	399	0.08	SLL	399	0.16	0.024	0.47	6.70
path-10M	FF	2	0.18	R	3	0.85	0.074	0.21	11.54
	LF	3	2.49	LLF	3	0.98	0.083	2.54	11.87
	SL	2	2.58	SLL	3	1.36	0.169	1.90	8.04

Table 3.7: Performance measurements for five classes of synthetically generated graphs: constant degree, rMat, 3D grid, clique chain and path. The column headings are equivalent to those in Table 3.6.

sidered. Every heuristic uses more than 300 colors, and the biggest difference between the number of colors used by any heuristic is less than 10.

The wiki-Talk and com-orkut graphs appear to benefit from ordering heuristics and illustrate what we believe is a coarse hierarchy of coloring quality in which FF $<$ R $<$ LLF $<$ LF $<$ SLL $<$ SL. On com-orkut, LLF produced a coloring of size 98, which was better than the 175 and 132 colors used by FF and R, respectively, but not as good as the 87 colors used by LF. In contrast, SLL nearly matched the superior coloring quality of SL, producing a coloring of size 84. On wiki-Talk, SLL produced a coloring of size 62, which was better than LF, LLF, R, and FF by a margin of between 8 to 40 colors, but not as good as SL, which used only 56 colors. These trends appear to exist, in general, for most of the graphs in the suite.

Scalability of JP-R, JP-LLF, and JP-SLL

The parallel performance of JP was measured by computing the speedup it achieved on 12 cores and by comparing the 1-core runtimes of JP to an optimized serial implementation of GREEDY. These results are summarized in Tables 3.6 and 3.7.

Overall, JP-LLF obtains a geometric-mean speedup — the ratio of the runtime on 1 core to the runtime on 12 cores — of 7.83 on the eight real-world graphs and 8.08 on the ten synthetic graphs. Similarly, JP-SLL obtains a geometric-mean speedup of 5.36 and 7.02 on the real-world and synthetic graphs, respectively.

Tables 3.6 and 3.7 also include scalability data for JP-FF, JP-LF, and JP-SL. Historically, JP-LF has been used with mixed success in practical parallel settings [129, 300, 179, 4]. Despite the fact that it offers little in terms of theoretical parallel performance guarantees, we have measured its parallel performance for our graph suite, and indeed JP-LF scales reasonably well: $\text{JP-LF}_1/\text{JP-LF}_{12} = 6.8$ as compared to $\text{JP-LLF}_1/\text{JP-LLF}_{12} = 8.0$ in geometric mean, not including cliqueChain400, which is omitted since JP-LF crashes due to excessive stack usage on cliqueChain400. The omission of cliqueChain400 highlights the dangers of using algorithms without good performance guarantees: it is difficult to know if the algorithm will behave badly given any particular input. In this respect, JP-FF is particularly vulnerable to adversarial inputs, as we can see by the fact that it crashes on europe-osm, which is not even intentionally adversarial. We also see this vulnerability with JP-SL, as well as generally poor scalability on the entire suite.

To measure the overheads introduced by using a parallel algorithm, the runtime T_1 of JP on 1 core was compared with the runtime T_S of an optimized implementation of GREEDY. This comparison was performed for each of the three parallel ordering heuristics we considered: R, LLF, and SLL. The serial runtime of GREEDY using FF is 2.5 times faster than JP-R on 1 core for the eight real-world graphs and 2.3 times faster on the ten synthetic graphs. We conjecture that GREEDY gains its advantage due to the spatial-locality advantage that results from processing the vertices in the linear order they appear in the graph representation. JP-LLF and JP-SLL on 1 core, however, are actually faster than GREEDY with LF and SL by 43.3% and 19% on the eight real-world graphs and 6% and 3% on the whole suite, respectively.

In order to validate that our implementation of GREEDY is a credible baseline, we compared it with a publicly available graph-coloring library, COLPACK [121], developed by Gebremedhin *et al.* and found that the two implementations appeared to achieve similar performance. For example, using the SL ordering heuristic, GREEDY is 19% faster than COLPACK in geometric-mean across the graph suite, though GREEDY is slower on 5 of the 16 graphs and as much 2.22 times slower for as-skitter.

3.7 Implementation techniques

This section describes the techniques we employed to implement JP and GREEDY for the evaluation in Section 3.6. We describe three techniques — join-trees [98], bit-vectors, and software prefetching — that improve the practical performance of JP. Where applicable, these same techniques were used to optimize the implementation of GREEDY. Overall, applying these techniques yielded a speedup of between 1.6 and 2.9 for JP and a speedup of between 1.2 and 1.6 for GREEDY on the rMat-G-2M, rMat-B-2M, web-Google, and as-skitter graphs used in Section 3.6.

Join trees for reducing memory contention

Although the theoretical analysis of JP in Section 3.2 does not concern itself with contention, the implementation of JP works to mitigate overheads due to contention. The pseudocode for JP in Figure 3-2 shows that each vertex u in the graph has an associated counter $u.counter$. Line 17 of JP-COLOR executes a JOIN operation on $u.counter$. Although Section 3.2 describes how JOIN can treat $u.counter$ as a join counter [76] and update $u.counter$ using an atomic decrement and fetch operation, the cache-coherence protocol [277] on the machine serializes such atomic operations, giving rise to potential memory contention. In particular, memory contention may harm the practical performance of JP on graphs with large-degree vertices.

Our implementation of JP mitigates overheads due to contention by replacing each join counter $u.counter$ with a join tree having $\Theta(|u.pred|)$ leaves. In particular, each join tree was sized such that an average of 64 predecessors of u map to each leaf through a hash function that maps predecessors to random leaves. We found that the join tree reduces T_1 for JP by a factor of 1.15 and reduces T_{12} for JP by between 1.1 and 1.3.

Bit vectors for assigning colors

To color vertices more efficiently, the implementation of JP uses vertex-local bit vectors to store information about the availability of low-numbered colors. Because JP assigns to each vertex the lowest-numbered available color, vertices tend to be colored with low-numbered colors. To take advantage of this observation, we store a 64-bit word per vertex u to track the colors in the range $\{1, 2, \ldots, 64\}$ that have already been assigned to a neighbor of u. The bit vector on $u.vec$ is computed as a "self-timed" OR reduction that occurs during updates on u's join tree. Effectively, as each predecessor v of u executes JOIN on u's join tree, if color(v) is in $\{1, 2, \ldots, 64\}$, then v OR's the word $2^{\text{color}(v)-1}$ into $u.vec$. When GET-COLOR(u) subsequently executes, GET-COLOR first scans for the lowest unset bit in $u.vec$ to find the minimum color in $\{1, 2, \ldots, 64\}$ not assigned to a neighbor of u. Only when no such color is available does GET-COLOR(u) scan its predecessors to assign a color to u.

We discovered that a large fraction of vertices in a graph can be colored efficiently using this practical optimization. We found that this optimization improved T_{12} for JP by a factor of 1.4 to 2.2, and a similar optimization sped up the implementation of GREEDY by a factor of 1.2 to 1.6.

Software prefetching

We used software prefetching to improve the latency of memory accesses in JP. In particular, JP uses software prefetching to mitigate the latency of the indirect memory access encountered when accessing the join trees of the successors of a vertex v on line 16 of JP-COLOR in Figure 3-2. This optimization improves T_{12} for JP by a factor of 1.2 to 1.5.

Interestingly, our implementation of GREEDY did not appear to benefit from using software prefetching in a similar context, specifically, to access the predecessors of a vertex

```
GREEDY-SD($G$)
let $G = (V, E)$
for $v \in V$
    $v.adjColors \leftarrow \emptyset$
    $v.adjUncolored \leftarrow \text{Adj}[v]$
    PUSHORADDKEY($v, Q[0][|v.adjUncolored|]$)
$s \leftarrow 0$
while $s \geq 0$
    $v \leftarrow$ POPORDELKEY($Q[s][\max \text{KEYS}(Q[s])]$)
    $v.color \leftarrow \min(\{1, 2, \ldots, |v.adjUncolored| + 1\} - v.adjColors)$
    for $u \in v.adjUncolored$
        REMOVEORDELKEY($u, Q[|u.adjColors|][|u.adjUncolored|]$)
        $u.adjColors \leftarrow u.adjColors \cup \{v.color\}$
        $u.adjUncolored \leftarrow u.adjUncolored - \{v\}$
        PUSHORADDKEY($u, Q[|u.adjColors|][|u.adjUncolored|]$)
        $s \leftarrow \max\{s, |u.adjColors|\}$
    while $s \geq 0$ and $Q[s] == \emptyset$
        $s \leftarrow s - 1$
```

Figure 3-5: The GREEDY-SD algorithm computes a coloring for the input graph $G = (V, E)$ using the SD heuristic. Each uncolored vertex $v \in V$ maintains a set $v.adjColors$ of colors used by its neighbors and a set $v.adjUncolored$ of uncolored neighbors of v. The PUSHORADDKEY method adds a specified key, if necessary, and then adds an element to that key's associated set. The POPORDELKEY and REMOVEORDELKEY methods remove an element from a specified key's associated set, deleting that key if the set becomes empty. The variable s maintains the maximum saturation degree of G.

on line 4 of GREEDY in Figure 3-1. We suspect that because GREEDY only reads the predecessors of a vertex on this line and does not write them, the processor hardware is able to generate many such reads in parallel, thereby mitigating the latency penalty introduced by cache misses.

3.8 The SD heuristic

Our experiments with serial heuristics detailed in the Appendix (Section 3.11) indicate that the SD heuristic tends to provide colorings with higher quality than the other heuristics we have considered, confirming similar findings by Gebremedhin and Manne [122]. Although we leave the problem of devising a good parallel algorithm for SD as an open question, we were able to devise a linear-time serial algorithm for the problem, despite conjectures in the literature [129, 75] that superlinear time is required. This section briefly describes our linear-time serial algorithm for SD.

Figure 3-5 gives pseudocode for the GREEDY-SD algorithm, which implements the SD heuristic. Rather than trying to define a priority function for SD, the figure gives the coloring algorithm GREEDY-SD itself, since the calculation of such a priority function would color the graph as a byproduct. At any moment during the execution of the algorithm,

	C						
Graph	FF	R	LF	ID	SL	SD	*Spark*
com-orkut	175	132	87	86	83	76	
liveJournal1	352	330	323	325	322	326	
europe-osm	5	5	4	4	3	3	
cit-Patents	17	21	14	14	13	12	
as-skitter	103	81	71	72	70	70	
wiki-Talk	102	85	72	57	56	51	
web-Google	44	44	45	45	44	44	
com-youtube	57	46	32	28	28	26	
constant1M	33	32	32	34	34	26	
constant500K	52	52	52	55	53	44	
graph500-5M	220	220	159	157	158	147	
graph500-2M	206	208	153	152	153	141	
rMat-ER-2M	12	12	11	11	11	8	
rMat-G-2M	27	27	15	15	15	11	
rMat-B-2M	105	105	67	67	67	59	
big3dgrid	4	7	7	4	7	5	
cliqueChain400	399	399	399	399	399	399	
path-10M	2	3	3	2	2	2	

Table 3.8: Performance measurements for six serial ordering heuristics used by GREEDY, where measurements for real-world graphs appear above the center line and those for synthetic graphs appear below. The "*Spark*" column contains bar graphs that pictorially represent the coloring quality for each of the ordering heuristics. The height of the bar for the coloring quality C_H of ordering heuristic H is proportional to C_H. Section 3.6 details the experimental setup and graph suite used.

the ***saturation degree*** of a vertex v as the number $|v.adjColors|$ of distinct colors of v's neighbors, and the ***effective degree*** of v as $|v.adjUncolored|$, its degree in the as yet uncolored graph.

The main loop of GREEDY-SD (lines 42–52) first removes a vertex v of maximum saturation degree from Q (line 43) and colors it (line 44). It then updates each uncolored neighbor $u \in v.adjUncolored$ of v (lines 45–50) in three steps. First, it removes u from Q (line 46). Next, it updates the set $u.adjUncolored$ of u's ***effective neighbors*** — u's uncolored neighbors in G — and the set $u.adjColors$ of colors used by u's neighbors (lines 47–48). Finally, it enqueues u in Q based on u's updated information (lines 49–50).

The crux of GREEDY-SD lies in the operation of the queue data structure Q, which is organized as an array of ***saturation tables***, each of which supports the three methods PUSHORADDKEY, POPORDELKEY, and REMOVEORDELKEY described in the caption of Figure 3-5. A saturation table can support these operations in $\Theta(1)$ time and allow its keys K to be read in $\Theta(K)$ time. At the start of each main loop iteration, entry $Q[i]$ stores the uncolored vertices in the graph with saturation degree i in a saturation table. The PUSHORADDKEY, POPORDELKEY, and REMOVEORDELKEY methods maintain the invariant that, for each table $Q[i]$, each key $j \in \text{KEYS}(Q[i])$ is associated with a nonempty set of vertices, such that each vertex $v \in Q[i][j]$ has saturation degree i and effective degree j.

	T_S						
Graph	FF	R	LF	ID	SL	SD	*Spark*
com-orkut	2.23	3.39	3.54	44.13	10.59	46.60	
liveJournal1	0.89	2.05	2.34	17.93	4.69	19.75	
europe-osm	1.32	13.36	17.15	48.59	19.87	52.73	
cit-Patents	0.50	1.62	2.00	9.82	3.21	10.08	
as-skitter	0.24	1.70	2.43	9.41	2.79	9.94	
wiki-Talk	0.09	0.35	0.49	2.79	0.61	2.90	
web-Google	0.09	0.22	0.25	1.68	0.47	1.77	
com-youtube	0.06	0.19	0.25	1.50	0.35	1.55	
constant1M	0.90	1.13	1.16	16.07	2.96	17.23	
constant500K	0.74	0.88	0.84	14.20	1.97	15.51	
graph500-5M	1.83	3.14	3.69	25.19	8.43	35.29	
graph500-2M	0.52	0.77	0.98	8.09	2.22	11.68	
rMat-ER-2M	0.47	0.93	1.07	10.10	2.22	9.13	
rMat-G-2M	0.48	0.92	1.18	9.17	2.59	9.07	
rMat-B-2M	0.50	0.83	1.00	8.44	2.41	8.64	
big3dgrid	0.41	3.34	4.07	13.61	4.77	15.30	
cliqueChain400	0.05	0.05	0.05	0.81	0.08	2.06	
path-10M	0.18	1.95	2.49	7.34	2.58	7.96	

Table 3.9: Performance measurements for six serial ordering heuristics used by GREEDY, where measurements for real-world graphs appear above the center line and those for synthetic graphs appear below. The "*Spark*" column contains bar graphs that pictorially represent the serial running time for each of the ordering heuristics. The height of the bar for the serial running time T_S of ordering heuristic H is proportional to the log of T_S. Section 3.6 details the experimental setup and graph suite used.

Theorem 24 GREEDY-SD *colors a graph* $G = (V, E)$ *according to the* SD *ordering heuristic in* $\Theta(V + E)$ *time.*

PROOF. PUSHORADDKEY, POPORDELKEY, and REMOVEORDELKEY operate in $\Theta(1)$ time, and a given saturation table's key set K can be read in $\Theta(K)$ time. Line 43 can thus find a vertex v with maximum saturation degree s in $\Theta(|\text{KEYS}(Q[s])|)$ time. Line 44 can color v in $\Theta(\deg(v))$ time, and lines 50–52 maintain s in $\Theta(s)$ time. Because $s + |\text{KEYS}(Q[s])| \leq \deg(v)$, lines 42–52 evaluate v in $\Theta(\deg(v))$ time. The handshaking lemma [77, p. 1172–3] implies the theorem, because each vertex in V is evaluated once. □

3.9 Related work

Parallel coloring algorithms have been explored extensively in the distributed computing domain [5, 229, 179, 131, 132, 201, 200, 15]. These algorithms are evaluated in the message-passing model, where nodes are allowed unlimited local computation and exchange messages through a sequence of synchronized rounds. Kuhn [200] and Barenboim and Elkin [15] independently developed $O(\Delta + \lg^* n)$-round message passing algorithms to compute a deterministic greedy coloring.

Several greedy coloring algorithms have been described in synchronous PRAM models. Goldberg *et al.* [131] describe an algorithm for finding a greedy coloring of $O(1)$-degree graphs in $O(\lg n)$ time in the EREW PRAM model using a linear number of processors. They observe that their technique can be applied recursively to color Δ-degree graphs in $O(\Delta \lg \Delta \lg n)$ time. Their strategy incurs $\Omega(\lg \Delta (V + E))$ (superlinear) work, however.

Catalyurek *et al.* [62] present the algorithm ITERATIVE, which first speculatively colors a graph G and then fixes coloring conflicts, that is, corrects the coloring where two adjacent vertices are assigned the same color. The process of fixing conflicting colors can introduce new conflicts, though the authors observe empirically that comparatively few iterations suffice to find a valid coloring. We ran ITERATIVE on our test system and found that JP-LLF uses 13% fewer colors and takes 19% less time in geometric mean of number of colors and relative time, respectively, over all graphs in our test suite. Furthermore, we found that JP-SLL uses 17% fewer colors, but executes in twice the time of ITERATIVE. We do not know the extent to which the optimizations enjoyed by our algorithms could be adopted by speculative-coloring algorithms, however, and so it is likely too soon to draw conclusions about comparisons between the strategies.

3.10 Conclusion

Because of the importance of graph coloring, considerable effort has been invested over the years to develop ordering heuristics for serial graph-coloring algorithms. For the traditional "serial" LF and SL ordering heuristics, we have developed "parallel" analogs — the LLF and SLL heuristics, respectively — which approximate the traditional orderings, generating colorings of comparable quality while offering provable guarantees on parallel scalability. The correspondence between serial ordering heuristics and their parallel analogs is fairly direct for LF and LLF . LLF colors any two vertices whose degrees differ by more than a factor of 2 in the same order as LF. In this sense, LLF can be viewed as a simple coarsening of the vertex ordering used by LF. Although SLL is inspired by SL, and both heuristics tend to color vertices of smaller degree later, the correspondence between SL and SLL is not as straightforward. We relied on empirical results to determine the degree to which SLL captures the salient properties of SL.

We had hoped that the coarsening strategy LLF and SLL embody would generalize to the other serial ordering heuristics, and we are disappointed that we have not yet been able to devise parallel analogs for the other ordering heuristics, and in particular, for SD. Because the SD heuristic appears to produce better colorings in practice than all of the other serial ordering heuristics, SD appears to capture an important phenomenon that the others miss.

The problem with applying the coarsening strategy to SD stems from the way that SD is defined. Because SD determines the order to color vertices while serially coloring the graph itself, it seems difficult to parallelize, and it is not clear how SD might correspond to a possible parallel analog. Thus, it remains an intriguing open question as to whether a parallel ordering heuristic exists that captures the same "insights" as SD while offering provable guarantees on scalability.

3.11 Appendix: Performance of serial ordering heuristics

Tables 3.8 and 3.9 summarizes our empirical evaluation of Greedy run on our suite of real-world and synthetic graphs using the six ordering heuristics from Section 3.1. The measurements were taken using the same machine and methodology as was used for Tables 3.6 and 3.7. As Tables 3.8 and 3.9 show, we found that, in order, FF, R, LF, SL, and SD generally produce better colorings at the cost of greater running times. ID was outperformed in both time and quality by SL. The figure indicates that LF tends to produce better colorings than FF and R at some performance cost, and SL produces better colorings than LF at additional cost. We found that SD produces the best colorings overall, at the cost of a 4.5 geometric-mean slowdown versus SL.

3.12 Acknowledgments

Thanks to Guy Blelloch of Carnegie Mellon University for sharing utility functions from his Problem Based Benchmark Suite with us [320]. Thanks to Aydın Buluç of Lawrence Berkeley Laboratory for helping us in our search for collections of large sparse graphs. Thanks to Mahantesh Halappanavar of Pacific Northwest National Laboratory for providing us with the code for Iterative [62]. Thanks to Assefaw Gebremedhin for input regarding the publicly available graph-coloring library ColPack [121]. Thanks to Jack Dennis of MIT CSAIL for helping us track down early work on parallel sorting and join counters. Thanks to Jeremy Fineman for helpful discussions on the amortized analysis of SD. Thanks to Angelina Lee and Justin Zhang of MIT CSAIL and Julian Shun and Harsha Vardhan Simhadri of Carnegie Mellon University for several helpful discussions.

Chapter 4

PARAD: A Work-Efficient Parallel Algorithm for Reverse-Mode Automatic Differentiation

This chapter presents PARAD the first work-efficient parallel algorithm for performing reverse-mode automatic differentiation. This work was conducted in collaboration with Tao B. Schardl, Brian Xie, Jie Chen, Aldo Pareja, Georgios Kollias, and Charles E. Leiserson.

Abstract

Automatic differentiation (AD) is a technique for computing the derivative of function $F : \mathbb{R}^n \to \mathbb{R}^m$ defined by a computer program. Modern applications of AD, such as machine learning, typically use AD to facilitate gradient-based optimization of an objective function for which $m \ll n$ (often $m = 1$). As a result, these applications typically use reverse (or adjoint) mode AD to compute the gradient of F efficiently, in time $\Theta(m \cdot T_1(F))$, where T_1 is the work (serial running time) of F. Although the serial running time of reverse-mode AD has a well known relationship to the total work of F, general-purpose reverse-mode AD has proven challenging to parallelize in a work-efficient and scalable fashion, as simple approaches tend to result in poor performance or scalability.

This chapter introduces PARAD, a work-efficient parallel algorithm for reverse-mode AD of determinacy-race-free recursive fork-join programs. We analyze the performance of PARAD using work/span analysis. Given a program F with work $T_1(F)$ and span (critical-path length) $T_\infty(F)$, PARAD performs reverse-mode AD of F in $O(m \cdot T_1(F))$ work and $O(\log m + \log(T_1(F))T_\infty(F))$ span. To the best of our knowledge, PARAD is the first parallel algorithm for performing reverse-mode AD that is both provably work-efficient and has span within a polylogarithmic factor of the original program F.

We implemented PARAD as an extension of Adept, a C++ library for performing reverse-mode AD for serial programs that is known for its efficiency. Our implementation supports the use of Cilk fork-join parallelism and requires no programmer annotations of parallel control flow. Instead, it uses compiler instrumentation to dynamically trace a program's series-parallel structure, which is used to automatically parallelize the gradient computation via reverse-mode AD. On eight machine-learning benchmarks, our implementation of PARAD achieves 1.5× geometric-mean multiplicative work overhead relative to the serial Adept tool, and 8.9× geometric-mean self-relative speedup on 18 cores.

4.1 Introduction

Automatic differentiation[1] [139, 280, 286, 193], or ***AD*** for short, aims to numerically compute the derivative of a function $F : \mathbb{R}^n \to \mathbb{R}^m$ defined by a computer program. Although AD has a long history of exploration and development in applications such as computational fluid dynamics, molecular dynamic simulations, engineering design optimization, sensitivity analysis, and uncertainty quantification, AD is commonly used today as a fundamental computational step in training neural networks for machine learning. In that context, the program is a neural network that defines a function F mapping a set of weights $W \in \mathbb{R}^n$ to a loss $L \in \mathbb{R}^m$, for which $m \ll n$ (often $m = 1$). AD is used when training the neural network to facilitate gradient-based optimization of L [44]. In contrast to symbolic [154] or numerical differentiation [56], AD provides an efficient way to compute partial derivatives for functions of many input variables, which makes AD appealing for training neural networks [18]. More broadly, efficient general-purpose AD sees a diversity of uses today, from general applications of AD in machine learning, such as in differentiable programming systems (e.g., [167, 46, 280]), to various applications in computational science.

For a function F computed serially in time $T_1(F)$, the gradient of F can be computed using either ***forward-mode*** AD [342], in time $\Theta(n \cdot T_1(F))$, or using ***reverse-mode*** (or ***adjoint-mode***) AD [230, 324], in time $\Theta(m \cdot T_1(F))$. Reverse-mode AD is therefore well suited for machine learning and has therefore grown in popularity.

This paper addresses the problem of automatically parallelizing general-purpose reverse-mode AD for programs implemented using ***(recursive) fork-join parallelism***, as supported by parallel programming languages including dialects of Cilk [116, 219, 172], Fortress [3], Kokkos [101], Habanero [16], Habanero-Java [61], Hood [42], HotSLAW [258], Java Fork/Join Framework [208], OpenMP [275, 13], Task Parallel Library [218], Threading Building Blocks (TBB) [291], and X10 [68]. Fork-join parallelism allows subroutines to be spawned recursively in parallel and iterations of parallel loops to execute concurrently. Fork-join programs expose fine-grained tasks that are allowed to execute in parallel, but are not required to. The execution and synchronization of fine-grained tasks is managed "under the covers" by a runtime system, which typically implements a randomized work-stealing scheduler [41, 116, 11, 38] to schedule and load-balance the computation among parallel ***worker threads***. Constructs such as **parallel for** can be implemented as syntactic sugar on top of the fork-join model. As long as a fork-join program contains no ***determinacy races*** [105] (also called ***general races*** [266]) — no cases where two logically parallel operations access the same memory location, and at least one access writes to the location — then it is ***deterministic***, meaning that every execution of the program on a given input performs the same operations, regardless of scheduling. Fork-join parallelism has emerged as a popular parallel-programming model that allows many programs to be implemented efficiently as deterministic parallel programs [33, 183, 317].

This paper explores the following problem: given a determinacy-race-free function $F : \mathbb{R}^n \to \mathbb{R}^m$ defined by a recursive fork-join parallel program, automatically parallelize the reverse-mode AD computation of F in a work-efficient and scalable manner that is efficient in practice. We have observed that, in practice, many applications of reverse-mode AD, including machine-learning applications, implement functions that satisfy these constraints.

General-purpose reverse-mode AD has long posed a challenge to parallelize efficiently [29], despite its substantial history of research and development (for a survey of previous

[1]Also known as algorithmic differentiation.

work, see [18]). Reverse-mode AD can be performed in parallel for the m dimensions of the output of F, but this approach yields minimal parallelism when m is small. Specialized algorithms have been developed to perform parallel reverse-mode AD for specific computations [164, 163, 162, 161], but these specialized approaches do not apply to recursive fork-join computations in general. Previously developed solutions [28, 301] to parallel reverse-mode AD either are not ***work efficient*** — the total computation involved is $\omega(m \cdot T_1(F))$ — or they suffer in parallel performance and scalability, for example, due to lock contention.

The challenges of parallelizing reverse-mode AD

To see the challenges in automatically parallelizing reverse-mode AD, let us first examine serial reverse-mode AD on a function F.

Intuitively, AD views the computation of F as a sequence of primitive arithmetic operations and primitive functions — such as addition, multiplication, sine, and cosine — and performs a nonstandard interpretation of F to calculate derivatives. Reverse-mode AD computes the derivative of F by applying the chain rule from differential calculus, starting from the outermost function, which is the last operation or function in the sequence. More precisely, let $L \in \mathbb{R}^m$ be the dependent variable computed by F, and let n be the length of the operation sequence to compute F. After at each step i, the reverse-mode AD algorithm has evaluated some suffix S_{n-i} of the computation of F, and it stores a set of ***gradients*** that encode the adjoint $\partial L/\partial W_{n-i}$, where W_{n-i} is the set of inputs to S_{n-i}. Step $i+1$ of the algorithm grows the evaluated suffix S_{n-i-1} from S_{n-i} by updating the set of gradients to store $\partial L/\partial W_{n-i-1} = (\partial L/\partial W_{n-i})(\partial W_{n-i}/\partial W_{n-i-1})$.

Reverse-mode AD is most commonly accomplished through maintenance of an auxiliary ***tape*** data-structure.[2] Conceptually, reverse-mode AD first performs an augmented execution of the function F, known as the ***forward pass***, to record data about F's computation onto the tape. After the forward pass completes, reverse-mode AD performs a ***reverse pass***, in which it applies the chain rule to the operations on the tape in reverse order. Section 4.2 describes an efficient serial reverse-mode AD algorithm in detail.

At a high level, the tape and the set of gradients pose two key challenges to parallelizing reverse-mode AD.

Parallelizing the tape Parallel reverse-mode AD must accommodate the parallelism in the computation of F. For a fork-join parallel program F, the spawning and synchronization of logically parallel tasks imposes a directed acyclic graph (DAG) of dependencies operations, rather than a sequence. Dependencies between primitive operations must be recorded efficiently in parallel during the forward-pass execution of F. In addition, the DAG structure of dependencies implies logical parallelism between operations in the reverse pass, which should be exploited to achieve performance and scalability.

Parallel maintenance of gradients As Section 4.2 describes, one well-known feature of reverse-mode AD is that a variable-read operation in the given function F corresponds to a write of a gradient value during the reverse pass of the reverse-mode AD computation of F, and vice versa [29, 301]. As a result, even if F is determinacy-race free, logically parallel read operations during the forward pass become logically parallel write operations during the reverse pass. Intuitive approaches to managing gradients can lead to poor performance.

[2]Also called a Wengert list [342].

For example, coordinating updates to gradients using locks can result in high contention that inhibits scalability. Alternatively, one might imagine maintaining gradients using P thread-local tables, where P is the number of processors executing the reverse pass. But a simple use of thread-local gradient tables is not work efficient, because reads of gradient values can occur asynchronously during a parallel execution of the reverse pass, and $O(P)$ work is needed to read a gradient out of the thread-local tables. Synchronizing these reads can reduce the total work, but at the cost of parallel scalability.

Previous approaches

Previous work [28] has explored parallel reverse-mode AD for OpenMP programs using ADOL-C [340], a library for reverse-mode AD in C++ programs. To accommodate OpenMP threads, a separate instance of ADOL-C is created for each thread, such that each thread operates on its own tape and set of gradients. Separate user code is invoked to combine gradient information from these parallel tapes at serial points in the computation. This thread-local AD approach can work efficiently for programs with simple parallel control flow, such as a sequence of parallel loops. But it is unclear how to generalize the approach to handle arbitrary recursive fork-join parallel programs while maintaining work efficiency and scalability. Nested parallel control flow in particular presents a significant challenge, because work efficiency can be precluded by the total work performed over the entire reverse pass to combine gradient information at nested synchronization points.

Previous work has also explored similar node-local approaches to parallelize reverse-mode AD for MPI programs [301], by assigning each MPI node to operate on a separate tape and replacing MPI communications in the forward pass with appropriate reversed communications to communicate gradient information in the reverse pass. These approaches are not work efficient [158], due to the cost of communicating and combining parallel gradient information.

PARAD: Work-efficient and scalable reverse-mode AD

This paper introduces **PARAD** a provably efficient parallel algorithm for reverse-mode AD for determinacy-race-free recursive fork-join programs. Given such a program F, PARAD computes the gradient of F using reverse-mode AD work-efficiently and with parallel scalability comparable to that of F itself. In particular, PARAD exploits the logical parallel control flow of the input of F to parallelize the reverse-mode AD computation of F.

In particular, Section 4.4 analyzes the parallel performance of PARAD using work/span analysis [77, Ch. 27]. The ***work*** of a computation is the total number of instructions executed, and the ***span*** is the length of a longest path of dependencies in the program. Section 4.4 shows that, given an input function $F : \mathbb{R}^n \to \mathbb{R}^m$ which takes work $T_1(F)$ and span $T_\infty(F)$ to compute, PARAD computes reverse-mode AD of F in work $\Theta(m \cdot T_1(F))$ and span $O(\log m + \log(T_1(F))T_\infty(F))$. To the best of our knowledge, PARAD is the first parallel algorithm for performing reverse-mode AD that both is provably work-efficient and has span within a polylogarithmic factor of $T_\infty(F)$.

To efficiently parallelize reverse-mode AD, PARAD implements an ***SP-Tape*** data structure, which records a tape for F efficiently in parallel, and a novel parallel algorithm for maintaining gradients.

Algorithm	T_s/T_1	T_s/T_{18}	T_1/T_{18}
PARAD	0.57	5.12	9.02
PARAD+S	0.66	5.88	8.89
Locks	0.45	2.77	6.14
Worker-Local	0.94	4.96	5.27

Figure 4-1: Performance comparison of PARAD, PARAD+S, Locks, and Worker-Local over the 8 application benchmarks discussed in Section 4.6. Average (geometric mean) work-efficiencies T_s/T_1 and 18-core speedups T_s/T_{18} are provided relative to the serial runtime T_s of Adept.

A work-efficient, scalable, deterministic parallel tape Section 4.3 describes the SP-Tape data structure for recording a tape of the forward-pass execution of F, including all series-parallel relationships between primitive operations and functions in F. The SP-Tape data structure ensures that, if F is determinacy-race free, then the SP-Tape recorded for the forward-pass execution of F is the same, regardless of how the forward-pass is scheduled at runtime. Section 4.3 justifies that the SP-Tape records the tape in a work-efficient and scalable manner with bounded contention. The maintenance of an SP-Tape resembles techniques in previous work for recording series-parallel dependencies in recursive fork-join programs [288], but with substantial extensions to implement reverse-mode AD.

Work-efficient parallel maintenance of gradients To achieve a work efficient and scalable reverse pass, PARAD uses a novel algorithm to maintain gradients that overcomes the problems of approaches based on locks or thread-local gradient tables. In contrast to lock-based approaches, the algorithm avoids the overheads of locks and bounds the contention involved in updating gradients. In contrast to approaches that use thread-local gradient tables, the algorithm allows gradients to be read in an asynchronous manner, while maintaining both work efficiency and span comparable to the input function F. Section 4.4 describes this algorithm for maintaining sets of gradients in parallel.

The LibPARAD parallel reverse-mode AD library

We implemented PARAD in a library, called LibPARAD, to evaluate the empirical performance of PARAD. LibPARAD extends the Adept [156] library for reverse-mode AD computation of serial C++ programs, which is known to exhibit low overheads in practice. LibPARAD supports the use of the Cilk programming language [116, 219, 172] to encode recursive fork-join parallelism. LibPARAD captures the series-parallel relationships between operations using compiler-inserted program instrumentation, based on a version of the CSI framework [302] that instruments the Tapir compiler intermediate representation of recursive fork-join parallelism [305]. This approach has the extra benefit that programmers need not annotate the logical parallelism in the program. Instead, LibPARAD captures this logical parallelism based on the linguistic constructs in the original program. Section 4.5 describes the implementation of LibPARAD as well as several practical optimizations that LibPARAD implements on top of the PARAD algorithm.

We evaluated LibPARAD's serial and parallel performance in practice on a variety of machine-learning benchmarks, and we compare the performance of LibPARAD to Adept applied to the serial projection [116, 305] of each benchmark, as well as to implementations that use fine-grained locks (Locks) and thread-local gradient tables (Worker-Local).

Figure 4-1 summarizes our empirical evaluation of LibPARAD in which we compared PARAD and PARAD+S, which incorporates additional optimizations described in Section 4.5, with algorithms using locks and worker-local tables. On average, the PARAD and PARAD+S algorithms have better scalability than Locks and Worker-Local both in terms of self-relative speedup and relative to the serial Adept code. Section 4.6 dives into the empirical evaluation of LibPARAD and examines how the scalability of each algorithms varies across benchmarks.

Contributions This paper makes the following contributions.

1. We introduce PARAD, an algorithm that performs reverse-mode AD for determinacy-race-free recursive fork-join parallel programs with substantially fewer overheads than existing systems. Given a determinacy-race-free recursive fork-join program $F : \mathbb{R}^n \to \mathbb{R}^m$, PARAD automatically parallelizes the reverse-mode AD computation of F.

2. Using work/span analysis [77, Ch. 27], we show that PARAD performs reverse-mode AD on a given function F in work $O(m \cdot T_1(F))$ and $O(\log m + \log(T_1(F))T_\infty(F))$. To the best of our knowledge, PARAD is the first parallel algorithm for performing reverse mode AD that is both provably work-efficient and has span within a polylogarithmic factor of $T_\infty(F)$.

3. We introduce LibPARAD, an implementation of PARAD for performing automatically parallel reverse-mode AD for determinacy-race-free recursive fork-join programs. LibPARAD extends the Adept [156] C++ library for serial reverse-mode AD, which is known to outperform other C++ AD libraries [325].

4. We study the empirical performance of LibPARAD on eight machine-learning benchmarks and compare the the PARAD, PARAD+S, Locks, and Worker-Local algorithms for reverse-mode AD.

Organization The remainder of the paper is organized as follows. Section 4.2 provides background on AD and fork-join parallelism. Section 4.3 describes the SP-Tape data structure and analyzes it in terms of work and span. Section 4.4 presents and analyzes PARAD's work-efficient and scalable reverse-mode algorithm. Section 4.5 describes the implementation of LibPARAD, a C++ library implementation of PARAD, based on the Adept C++ library for serial reverse-mode AD. Section 4.6 compares the empirical performance of LibPARAD against Adept and a lock-based implementation of reverse-mode AD. Section 4.7 discusses related work in parallel AD. Section 4.8 provides concluding remarks.

4.2 Preliminaries

This section provides background information on the serial algorithm for reverse-mode AD, recursive fork-join parallelism, the dag model of multithreading, and work/span analysis. To simplify the description of reverse-mode AD, we shall consider input programs $F : \mathbb{R}^n \to \mathbb{R}^m$ for which $m = 1$. It is straightforward to extend this description for programs where $m > 1$.

A serial algorithm for reverse-mode AD

In general, a serial reverse-mode AD computation, as implemented by tools including Adept [156], ADOL-C [340], and Tapenade [145], operates in two passes. Given a function

Forward pass program

```
TwoByTwoMatVecSqLoss($a, b, c, d, e, f$):
1  $g = a \cdot e + b \cdot f$
2  $h = c \cdot e + d \cdot f$
3  $L = g^2 + h^2$
4  return $L$
```

Gradient table state after each step of reverse-mode AD

Step	∂a	∂b	∂c	∂d	∂e	∂f	∂g	∂h	∂L
0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	$2g$	$2h$	0
2	0	0	$2h \cdot e$	$2h \cdot f$	$2h \cdot c$	$2h \cdot d$	$2g$	0	0
3	$2g \cdot e$	$2g \cdot f$	$2h \cdot e$	$2h \cdot f$	$2h \cdot c + 2g \cdot a$	$2h \cdot d + 2g \cdot b$	0	0	0

Statement Stack

	index	end index
S_0	$index(g)$	4
S_1	$index(h)$	8
S_2	$index(L)$	10

Operation Stack

	index	mul
O_0	$index(a)$	$\partial g/\partial a = e$
O_1	$index(e)$	$\partial g/\partial e = a$
O_2	$index(b)$	$\partial g/\partial b = f$
O_3	$index(f)$	$\partial g/\partial f = b$
O_4	$index(c)$	$\partial h/\partial c = e$
O_5	$index(e)$	$\partial h/\partial e = c$
O_6	$index(d)$	$\partial h/\partial d = f$
O_7	$index(f)$	$\partial h/\partial f = d$
O_8	$index(g)$	$\partial L/\partial g = 2g$
O_9	$index(h)$	$\partial L/\partial h = 2h$

Figure 4-2: Illustration of a serial reverse-mode AD computation on simple program, TwoByTwoMatVecSqLoss. The tables in the top right show the statement and operation stacks, recorded during the execution of the forward pass of the program. The bottom table shows the state of the gradient table after processing each statement (in reverse order) during the reverse pass.

```
SerialReversePass($S, O, G$):
1  $i \leftarrow |O|-1$
2  for $k = |S|-1, |S|-2, \ldots, 0$
3      $\alpha \leftarrow G[S_k.index]$
4      $G[S_k.index] \leftarrow 0$
5      while $i > S_{k-1}.endIndex$
6          $G[O_i.index] \mathrel{+}= \alpha \cdot O_i.mul$
7          $i \mathrel{-}= 1$
```

Figure 4-3: Pseudocode for the serial algorithm for computing the reverse pass over the statement stack S and operation stack O with input gradients in the table G.

$F : \mathbb{R}^n \to \mathbb{R}^m$, the forward pass first executes F and records the primitive operations of F onto a tape data structure. The reverse pass maintains a table of gradient values and processes the tape data structure step by step in reverse, updating the gradient values at each step.

To examine the algorithm more closely, let us first examine the tape data structure and the forward pass. The serial tape data structure consists of two stacks: a ***statement stack***, corresponding to writes to variables in the program F, and an ***operation stack***, that records the values read to compute statements. Each differentiable variable v in the program, corresponding to an executed statement in F, is assigned a unique integer identifier $index(v)$, called the ***gradient index***. A statement-stack entry contains the index associated with the left-hand-side variable v, and the length of the operation stack when the statement was inserted. Each operation-stack entry contains the index of the variable u being read, as well as the partial derivative of the statement's left-hand-side variable v with respect to u. Figure 4-2 illustrates the statement and operation stacks recorded for a simple program with 3 statements.

To see how the forward pass populates the statement and operation stacks, consider the forward-pass executing of TwoByTwoMatVecSqLoss in Figure 4-2 line by line. Line 1 computes the statement $g = a \cdot e + b \cdot f$. The expression on the right-hand side of the statement is evaluated first and the operations O_0, O_1, O_2, O_3 are pushed onto the operation stack. After the right-hand expression is evaluated, an entry S_0 is pushed onto the statement stack recording the gradient index $index(g)$ of the left-hand-side variable g, and the current length 4 of the operation stack. As a result, after line 1 is executed, statement S_0 on the statement stack identifies operations O_0, O_1, O_2, and O_3 as storing partial derivatives $\partial g/\partial a$, $\partial g/\partial e$, $\partial g/\partial b$, and $\partial g/\partial f$, respectively. Line 2 is handled similarly by pushing operations O_4, O_5, O_6, O_7 to the operation stack and then pushing statement S_1 to the statement stack. Lastly, to handle line 3, operations O_8, O_9 are pushed onto the operation stack and statement S_2 onto the statement stack.

After executing the function, the reverse pass creates a ***gradient table*** with one entry for each gradient index created during the execution of the forward pass. Initially, all entries of the gradient table are set to 0, except for (one or more) variables whose derivatives are already known. In the example in Figure 4-2, there is a single loss variable L whose gradient index is initialized to 1.

The gradient-table state after each step of the reverse-mode AD algorithm is presented in Figure 4-2. In the figure, each row presents the state of the gradient table after a step of the reverse pass. Each row therefore encodes the coefficients of a differential expression obtained via differentiation of an implicit function. Row 0, for example, encodes the initial (trivial) differential expression $\partial L = 1 \cdot \partial L$. After processing statement S_2, this expression is transformed to $\partial L = (\partial L/\partial g)\partial g + (\partial L/\partial h)\partial h = 2g\ \partial g + 2h\ \partial h$. The gradient table on Row 1 encodes this transformation, in which the coefficient of ∂L is 0, and the coefficients of ∂g and ∂h are $2g$ and $2h$ respectively. To process statement S_1 the reverse pass reads the relationship between ∂h and the operations used to compute h encoded on the operation stack: $\partial h = c\ \partial e + e\ \partial c + d\ \partial f + f\ \partial d$. The reverse pass then uses this relationship to update the differential expression encoded in the gradient table. The new differential expression, encoded on Row 2, is $\partial L = 2g\ \partial g + 2h(c\ \partial e + e\ \partial c + d\ \partial f + f\ \partial d)$. The last step processes statement S_0 as we processed S_1, and results in the final gradient table (Row 3) that contains the partial derivatives of L with respect to each variable in the program.

Figure 4-3 gives psuedocode for the SerialReversePass procedure that performs the reverse pass, specifically, the updates to the gradient table using a previously recorded statement stack S, operation stack O, and input gradients G. The total work to maintain the two stacks and execute SerialReversePass is $\Theta(|O|) + \Theta(|S|) = \Theta(T_1(F))$ for an input program F with work $T_1(F)$.

Observations Conceptually, PARAD makes use of two observations of this serial reverse-mode AD algorithm to parallelize it. First, during the forward-execution of a parallel program, the serial tape data structure behaves like a list-monoid with an append operator. Second, the read and write sets of the forward-execution are swapped in the reverse-pass over the tape, i.e., every read becomes a write, and every write becomes a read. PARAD leverages these observations in its design of the SP-tape for recording the forward pass, and in its algorithm for automatically parallelizing the reverse pass to compute derivatives.

Recursive fork-join parallelism

Recursive fork-join parallelism allows logical parallelism in a program to be exposed using the keywords [77, Ch. 27] **spawn**, **sync**, and **parallel for**. When preceding a function call F, the **spawn** keyword ***spawns*** F, allowing F to execute in parallel with its ***continuation*** — the statement immediately after the spawn of F. The **sync** keyword complements the **spawn** keyword and acts as a local barrier that joins together, or ***syncs***, the parallelism specified by **spawn**. When a function F reaches a **sync**, control is not allowed to pass that **sync** until all functions spawned previously in F return. These keywords can be used to implement other parallel control constructs, such as the **parallel for** loop, which allows all of its iterations to operate logically in parallel.

A recursive fork-join program has a ***serial projection*** [116, 305], which intuitively is the serial program derived by removing parallel keywords from the fork-join program. If the fork-join program contains no determinacy races, then every execution of the program matches that of its serial projection.

The computation dag model

An execution of a fork-join program can be modeled as a computation dag $G = (V, E)$. Each directed edge represents a ***strand***, that is, a sequence of executed instructions with no **spawn** or **sync** statements. The execution of a **spawn** statement results in a ***spawn vertex***, which contains two successor strands. The execution of a **sync** statement results in a ***sync vertex***, which contains multiple incoming edges.

The dag G is a ***series-parallel dag*** [105], which means that G has two distinguished vertices — a ***source*** vertex, from which one can reach every other vertex in G, and a ***sink*** vertex, which is reachable from every other vertex in G — and can be constructed by recursively combining pairs of series-parallel dags using series and parallel combinations. A ***series combination*** combines two dags G_1 and G_2 by identifying the sink vertex of G_1 with the source vertex of G_2. A ***parallel combination*** combines two dags G_1 and G_2 by identifying their source vertices with each other and their sink vertices with each other.

The recursive construction of a series-parallel dag can be represented as a binary tree, called the ***SP tree*** [105], as follows. Each leaf in the SP tree represents a strand in the computation dag, and each internal node is either an S-node or a P-node. A subtree of the SP tree represents a series-parallel subdag of the computation dag. An S-node represents a series composition of the two subdags represented by its children. A P-node represents a parallel composition of the two subdags represented by its children.

Work/span analysis

Given a fork-join program whose execution is modeled as a DAG A, we can bound the P-processor running time $T_P(A)$ of the program using ***work/span analysis*** [77, Ch. 27]. The work $T_1(A)$ is the number of instructions in A, and the span $T_\infty(A)$ is the length of a longest path in A. Greedy schedulers [49, 99, 137] can execute a deterministic program with work T_1 and span T_∞ on P processors in time T_P satisfying $\max\{T_1/P, T_\infty\} \leq T_p \leq T_1/P + T_\infty$. A similar bound can be achieved by more practical work-stealing schedulers [40, 41]. The ***speedup*** of an algorithm on P processors is T_1/T_P, which the inequality shows to be at most P in theory. The ***parallelism*** T_1/T_∞ is the greatest theoretical speedup possible for any number of processors.

4.3 The SPTape Data Structure

This section describes the SPTape data structure that PARAD uses to record, in parallel, the statement and operation stacks for reverse-mode AD, and the series-parallel dependencies in the program. After recording, the SPTape supports parallel traversals with work and span proportional to that of the original recorded program. For later use, we describe and analyze generic parallel traversal algorithms over the SPTape and provide a set of rules governing memory access during the traversal that ensure the absence of determinacy races.

For didactic simplicity, we describe the SPTape data structure for *binary* recursive fork-join programs, in which each node in the computation dag has at most two incoming edges.

Basic structure of an SPTape

The SPTape data structure for a recursive fork-join program stores an SP tree [105] of the program augmented with ***data nodes*** that store "subtapes." A ***subtape*** contains a statement and operation stack that are used to record derivative dependencies within a strand. The subtapes in the SPTape represent an ordered partitioning of the statement and operation stack data structures employed by the serial AD tool discussed in Section 4.2.

Recording an SPTape serially

An SPTape is constructed incrementally during the execution of the forward pass. We shall first see a serial algorithm for constructing the SP-Tape, and then we shall see how to parallelize this algorithm.

To record an SPTape, a single processor maintains a ***shadow stack*** that is updated based on the parallel control flow of the forward-pass execution. Each entry on the shadow stack stores a local SPTape T, which is initialized with a single root node. When an entry is popped from the shadow stack, the local SPTape T for the popped entry is appended to the children of the SPTape node of the new top entry on the stack.

Figure 4-4 presents pseudocode for creating an SPTape. At a high level, this pseudocode creates S and P nodes in the SPTape based on **spawn** and **sync** statements in the program, and it records derivative dependencies in the subtape stored in the data node at the top of the shadow stack. Figure 4-5 illustrates an example of how the operations on the shadow stack execute in a simple example fork-join program. As the example shows, a P-node is

```
PushShadow($K$, $type$)
1  Push($K$)
2  Top($K$).$T.type \leftarrow type$

PopShadow($K$)
1  $\tau \leftarrow$ Top($K$).$T$
2  Pop($K$)
3  Append(Top($K$).$T.children$, $\tau$)
```

```
On entering a function:

1  PushShadow($K$, $Series$)
Before executing arithmetic:

1  if Top($K$).$T.type \neq Data$
2      PushShadow($K$, $Data$)
On returning from a function:

1  if Top($K$).$T.type == Data$
2      PopShadow($K$)
3  PopShadow($K$)
Immediately after executing a sync:

1  PopShadow($K$)
```

```
On executing a spawn:

1  PushShadow($K$, $Parallel$)
On executing the continuation of a spawn:

1  PushShadow($K$, $Series$)

Immediately before executing a sync:

1  if Top($K$).$T.type == Data$
2      PopShadow($K$)
3  PopShadow($K$)
```

```
Combine($K$, $\tau$)
1  Append(Top($K$).$T.children$, $\tau$)
```

Figure 4-4: Pseudocode for the maintenance of the SPTape data structure. The variable K denotes the shadow stack, each entry of which contains a local SP-Tape T. The field $T.type$ identifies the type of the root node of T. The field $T.children$ is a list of children of the root node of T. The Combine method is used to incorporate an SPTape τ recorded in parallel.

pushed onto the stack when a program executes a **spawn**, reflecting the fact that both the spawned function and its continuation can execute in parallel. S-nodes are pushed onto the stack when the program enters a function or a continuation of a **spawn**. Nodes are popped off the stack at the ends of functions and around **sync** operations.

The following lemma shows that, when the serial execution of a series-parallel computation dag G completes, the local SPTape at the top of the shadow stack records the gradient information and series-parallel dependencies for G.

Lemma 25 *When a serial execution completes a computation dag G, the top of the shadow stack stores a local* SPTape *that records the execution of G.*

Proof. The proof follows by induction on the structure of G. In the base case, G is a single strand u, and the top of the shadow stack stores an SPTape with a single S-node containing a single child data node that records the derivative dependencies in u. Otherwise G is the result of a series or parallel composition of subdags G_1 and G_2. In either case, the pseudocode in Figure 4-4 ensures that an entry for G is pushed onto the shadow stack at the beginning of G, and separate pushes and pops occur at the beginning and end, respectively, of each subdag. When popping the shadow stack at the end of each subdag, PopShadow

```
TwoByTwoMatVecSqLoss($a, b, c, d, e, f, K$):
PushShadow($K$, Series)
PushShadow($K$, Parallel)
spawn $\lambda${
    PushShadow($K$, Series)
    PushShadow($K$, Data) // ($D_1$)
    $g = a \cdot e + b \cdot f$
    PopShadow($K$) // ($D_1$)
    PopShadow($K$) // Series
}
PushShadow($K$, Series)
PushShadow($K$, Data) // ($D_2$)
$h = c \cdot e + d \cdot f$
PopShadow($K$) // ($D_2$)
PopShadow($K$) // Series
sync
PopShadow($K$) // Parallel
PushShadow($K$, Data) // ($D_3$)
$L = g^2 + h^2$
PopShadow($K$) // ($D_3$)
PopShadow($K$) // Series
return $L$
```

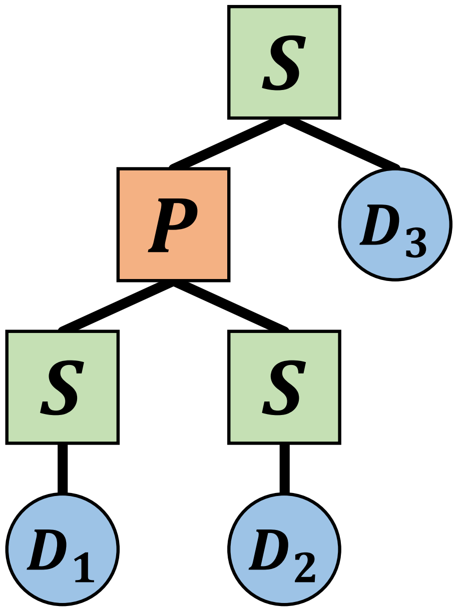


Figure 4-5: An example of the SPTape recorded for a parallel implementation of TwoByTwoMatVecSqLoss.

in Figure 4-4 shows that the new top of the shadow stack is either a P-node, if a spawn executed, or an S-node otherwise. In addition, Figure 4-4 shows that the local SPTape formerly at the top of the stack is appended to the list of children for the new top of the stack. Hence the top of the shadow stack stores a node of the correct type and correct child SPTape structures for the execution of G_1 and G_2. □

Parallelizing SPTape construction

To construct an SPTape in parallel, the parallel execution of a computation dag G is augmented to maintain separate shadow stacks. Intuitively, consider two series-parallel subdags G_1 and G_2 of G that are composed in parallel and are scheduled to execute in parallel. The execution uses distinct shadow stacks to separately record the SPTape structures of G_1 and G_2 and then combines those SPTape structures when execution reaches the common sink vertex of G_1 and G_2. In modern dialects of Cilk, this behavior can be accomplished using reducer hyperobjects [115]. We describe this behavior generically for series-parallel computation dags.

We model the scheduling of a computation dag G as a partitioning of the strands into ***scheduling components***, such that logically parallel strands execute in parallel if the scheduler places the strands into distinct scheduling components. We assume that the scheduler maintains the following properties of scheduling components:

- A new scheduling component can only begin at a successor strand u of a spawn vertex in G.

```
WalkSPTape($node$, $G$, VisitSubTape, $dir$):
 1  if $node.type = D$:
 2      VisitSubTape($node.S$, $node.O$, $G$, $dir$)
 3  $C = node.children$
 4  If $dir$ is right first reverse $C$.
 5  if $node.type = S$:
 6      for $c \in C$
 7          RightFirstTraversal($c$, $G$, $dir$)
 8  if $node.type = P$:
 9      parallel for $c \in C$:
10          RightFirstTraversal($c$, $G$, $dir$)
```

Figure 4-6: Pseudocode for the parallel right-first traversal of SPTape.

- Consider a spawn vertex s, which is the source vertex of a series-parallel subdag G produced via parallel composition. A new component C that begins at a successor strand u of s terminates at a predecessor strand v of the sink vertex of G such that v is reachable from u.

These properties create a correspondence between scheduling components and series-parallel subdags of G. In particular, the same scheduling component contains both the first strand in a series-parallel subdag and the last strand in that subdag. We shall also assume that the execution of strands within the same component follows a depth-first traversal consistent with the execution of the serial projection of that computation. Practical work-stealing schedulers [40, 41] typically satisfy these assumptions.

The parallel execution of the computation dag is augmented to maintain separate SPTape structures for distinct scheduling components. At the first strand s of each scheduling component C, the scheduler creates a new local SPTape for C. Subsequent operations within C operate on C's local SPTape as described in Figure 4-4. At the end of C, the local SPTape τ for C is combined into the shadow stack K' of the scheduling component C' that contains the predecessor w of s. In particular, τ is appended to the list of children of the root note of the SPTape at the top of K', using the Combine method in Figure 4-4. The following lemma extends Lemma 25 to incorporate combining local SPTape structures.

Lemma 26 *When execution completes a computation dag G, the top of the shadow stack stores a local* SPTape *that records the execution of G.*

Proof. The proof follows by extending the induction in Lemma 25 to handle computation dags G resulting from a composition involving a subdag G' whose strands belong to a different scheduling component. Suppose the lemma holds for such a subdag G'. Because the initial strand of G' must be a successor of a spawn vertex, G must be the result of a parallel composition. Hence, the pseudocode in Figure 4-4 shows that, at the end of executing G, the top of the shadow stack contains an SPTape structure T rooted at a P-node. The Combine routine appends the SPTape τ for G' to the children of this P-node, yielding an SPTape T that correctly represents G as a parallel composition involving G'.
□

Race-free traversals of SPTape

The PARAD algorithm makes use of parallel traversals over the SPTape. Here we provide conditions on which these traversals are race-free[3], and analyze their work and span.

For didactic purposes, consider a table G_x mapping gradient indices to distinct memory locations. Consider a call to VisitSubTape(S, O, G, dir, G_x), and let I_S, I_O be the sets of gradient indices appearing in the statement stack S and operation stack O respectively. We say a parallel traversal of an SP-Tape obeys the ***safe adjoint access property*** if VisitSubTape performs a read of $G_x[gid]$ only if $gid \in I_S \cup I_O$, and only performs a write to $G_x[gid]$ if $gid \in I_S$.

As Lemma 27 shows below, a parallel traversal of an SP-Tape produced by a determinacy-race-free program is free of data-races on entries of G_x if the traversal has the safe adjoint access property.

Lemma 27 *Let T be an* SPTape *produced by a determinacy-race-free fork-join program P. Consider a parallel traversal of T that obeys the safe adjoint access property. Then, the parallel traversal of T has no data races on accesses to tables G_x indexed by gradient indices appearing in statements and operations on the tape.*

Proof. Consider a parallel traversal of an SPTape T that has the safe adjoint access property. We will prove that if there exists a data race on entries of G_x during the traversal of T, then there exists a determinacy race in the program P that produced T.

Consider subtapes U, V corresponding to the two strands u, v in the program P. By Lemma 26, the subtapes U, V will be processed in parallel by WalkSPTape if and only if their corresponding strands u, v were logically in parallel in P.

Let $I_S(U)$ and $I_S(V)$ be the sets of gradient indices appearing in statements on the subtapes U and V respectively. Similarly define sets $I_O(U)$ and $I_O(V)$ for gradient indices appearing on the operation stacks.

Suppose there exists a data race on $G_x[gid]$ when evaluating VisitSubTape on U and V in parallel. For the data race to exist, at least one of these invocations must perform a write to $G_x[gid]$. Suppose the invocation processing U performs a write to $G_x[gid]$. Then $gid \in I_S(U)$, since VisitSubTape has the safe adjoint access property. The processing of V has either a write or a read to $G_x[gid]$ which implies (by safe adjoint access property) that $gid \in (I_S(V) \cup I_O(V))$. Since each statement in $U.S, V.S$ corresponds to a write performed in the original program P, and each operation $U.O, V.O$ correspond to reads in P, there exists a determinacy race between strands u and v in the original program P. □

Work/span analysis of SPTape traversal

Subsequent algorithms will perform parallel traversals over the SPTape data structure. Here we prove bounds on the work and span of a traversal whose VisitSubTape evaluates a function $f(\cdot)$ on each operation and statement stack entry which performs $O(\sigma)$ amortized work (amortized over the whole traversal), and has worst-case span $O(\upsilon)$.

Lemma 28 *Consider a parallel traversal by* WalkSPTape *of a* SPTape *T produced by a parallel program P with work $T_1(P)$ and span $T_\infty(P)$. If* VisitSubTape *processes each statement and operation with a function $f(\cdot)$ that performs $O(\sigma)$ amortized work (amortized*

[3]These conditions assume that the recorded program was, itself, free of determinacy races.

over the whole traversal) and has $O(\upsilon)$ worst-case span, then WalkSPTape *performs* $T_1 = O(\sigma T_1(P))$ *work and has* $T_\infty = O(\upsilon T_\infty(P))$ *span.*

Proof. The sum of the statement and operation stack sizes in the SPTape T is bounded by the total work of the program P. The total work of WalkSPTape is, therefore, $O(\sigma T_1(P))$. By Lemma 26, there is a one-to-one correspondence between strands in P and subtapes in T, and two subtapes in T are logically in parallel if and only if their associated strands are logically in parallel in P. Consider the sequence of strands $(u_1, u_2, \ldots, u_k)$ that forms the critical path in P. The corresponding set of subtapes in T for the critical path can be found $(U_1, U_2, \ldots, U_k)$. Uniformly scaling the work of all instructions in P by υ does not alter the critical path and scales the span by υ. The span of WalkSPTape when applying a function $f(\cdot)$ with worst-case span υ is, therefore, at most $T_\infty = O(\upsilon T_\infty(P))$ □

4.4 The PARAD algorithm

This section presents PARAD, a work-efficient algorithm for performing parallel reverse-mode AD. For a determinacy-race-free fork-join program F with work $T_1(F)$ and span $T_\infty(F)$, the program R that performs reverse-mode AD on F has work $T_1(R) = \Theta(m \cdot T_1(F))$ and span $T_\infty(R) = \Theta(\log m + T_\infty(F) \log(T_1(F)))$.

Design of the PARAD algorithm

The PARAD algorithm is designed to parallelize the serial reverse-mode AD algorithm based upon the series-parallel structure of the recorded program. The serial reverse-mode AD algorithm can be implemented by executing the SerialReversePass function on each subtape in the order of a right-first traversal of the SPTape. The problem with parallelizing this traversal, however, is the presence of write-write races on the gradient table, which occur when the original program read the same memory location in-parallel. The key challenge solved by PARAD is the resolution of these races in a work-efficient and scalable manner.

To resolve write-write races on the gradient table, PARAD employs the strategy to precompute a unique memory location where each operation can safely deposit its gradient contribution. These memory locations are assigned from a ***deposit array*** that provides one slot for each recorded operation. The gradient contributions are aggregated when a statement extracts its gradient value. In the serial AD algorithm, a statement can extract its gradient value by reading from the global gradient table. In PARAD, however, a statement must potentially collect and sum multiple gradient contributions. To allow this aggregation to be performed efficiently, the deposit array is organized so that all the gradient contributions a statement must collect are contiguous in the deposit array.

Let us discuss the structure of the PARAD algorithm whose pseudocode is provided in Algorithm 1.

The organization of the deposit array in PARAD is accomplished as follows. Step 1 of PARAD computes a table O_S that associate each operation with the statement that will consume its gradient contribution. Next, in Step 2 PARAD collects and semisorts all operations o by their associated statement $O_S[o]$[4]. The semisorted array of operations O^* is used to assign each operation a unique index in the deposit array D based upon its location in O^*. The assignment of operations to deposit array locations is recorded in a table O_{snd}.

[4]For didactic simplicity, the pseudocode of PARAD additionally semisorts by each operation's gradient index to make it simpler to export them to a global gradient table at the end of the computation.

Algorithm 1 PARAD Algorithm

Inputs: Gradient table G, and SP-Tape T.
Outputs: Updated gradient table G.

```
1. Construct $O_S$ mapping operations to statements.
    • Perform a parallel left-first traversal of the SP-Tree. At statement $s$, set $G_S[s.gid] = s$.
      At operation $o$, set $O_S[o] = G_S[o.gid]$.
2. Construct $O_{snd}$ and $S_{rcv}$.
    • Traverse the SP-Tape $T$ and pack all operations into contiguous array $O^*$.
    • Semisort $O^*$ using the key function $k_1(o) = o.gid$.
    • For each subarray of $O^*$ with operations of equal $k_1(o)$, semisort using the key
      $k_2(o) = O_S[o]$.
    • For each subarray $O^*[m..m+k]$ of operations $o$ with equal $k_2(o)$ do the following. Set
      $S_{rcv}[k_2(o)] = (m, m+k)$. For $l = m, m+1, \ldots, m+k$ set $O_{snd}[O^*[l]] = l$.
3. Perform reverse-mode AD.
    • Allocate deposit array $D$ of size $|O^*|$.
    • Perform a right-first traversal.
    • At statement $s$, let $(m, m+k) = S_{rcv}[s]$ and compute
      $\alpha = G[s.gid] + sum(D[m..m+k])$. Set $G[s.gid] = 0$, and $D[m..m+k] = 0$.
    • At operation $o$, compute $\beta = \alpha \cdot o.mul$ and set $D[O_{snd}[o]] = \beta$.
4. Export gradients in $D$ to gradient table $G$.
    • Find subarrays $O^*[n..n+r]$ in $O^*$ of operations $o$ with equal gradient index (i.e.
      $k_1(o) = o.gid$).
    • For each sub subarray, set $G[O^*[n].gid] = sum(D[n..n+r])$.
```

Since all operations associated with the same statement are contiguous in O^*, the gradient contributions for a statement are in a contiguous range of the deposit array D. A table S_{rcv} records, for each statement s, a reference to the subarray $D[m..m+k]$ of k gradient contributions to s.

The deposit array is now used by PARAD to avoid write-write races during its right-first traversal of the SPTape. Step 3 of PARAD performs reverse-mode AD via a right-first traversal of the SPTape. The final gradients are then exported in Step 4 to a global gradient table for application-specific use (e.g., for a gradient descent step).

Figure 4-7 provides an illustration of the deposit array structure for the parallelized TwoByTwoMatVecSqLoss function. The tables O_S, S_{rcv}, and O_{snd} are shown in Figure 4-7 as columns of the tables illustrating the statement and operation stacks. The deposit array structure is provided after processing each of the subtapes D_3, D_2, and D_1.

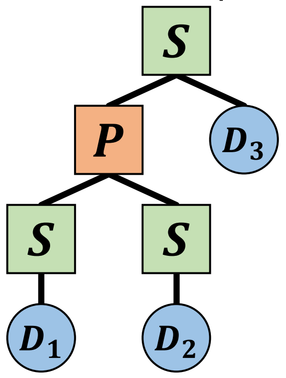
Theorem 29 PARAD *and* SerialReversePass *compute the same gradients for a recorded program P that is free of determinacy races.*

Proof. We first compare a serial execution of PARAD to the execution of SerialReversePass. In an execution of SerialReversePass, each operation accumulates its gradient contribution into the gradient table G. This accumulated value is extracted when

Statements

		index	endindex	S_{rcv}
D_1	S_0	index(g)	4	d_8
D_2	S_1	index(h)	8	d_9
D_3	S_2	index(L)	10	*input*

SP Tape



Operations

		index	mul	O_S	O_{snd}
D_1	O_0	index(a)	$\frac{\partial g}{\partial a} = e$	*output*	d_0
	O_1	index(e)	$\frac{\partial g}{\partial e} = a$	*output*	d_1
	O_2	index(b)	$\frac{\partial g}{\partial b} = f$	*output*	d_3
	O_3	index(f)	$\frac{\partial g}{\partial f} = b$	*output*	d_4
D_2	O_4	index(c)	$\frac{\partial h}{\partial c} = e$	*output*	d_6
	O_5	index(e)	$\frac{\partial h}{\partial e} = c$	*output*	d_2
	O_6	index(d)	$\frac{\partial h}{\partial d} = f$	*output*	d_7
	O_7	index(f)	$\frac{\partial h}{\partial f} = d$	*output*	d_5
D_3	O_8	index(g)	$\frac{\partial L}{\partial g} = 2g$	S_0	d_8
	O_9	index(h)	$\frac{\partial L}{\partial h} = 2h$	S_1	d_9

Evolution of deposit array during PARAD

step	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
	0	0	0	0	0	0	0	0	0	0
D_3	0	0	0	0	0	0	0	0	2g	2h
D_2	0	0	$2h \cdot c$	0	0	$2h \cdot d$	$2h \cdot e$	$2h \cdot f$	2g	0
D_1	$2g \cdot e$	$2g \cdot a$	$2h \cdot c$	$2g \cdot f$	$2g \cdot b$	$2h \cdot d$	$2h \cdot e$	$2h \cdot f$	0	0
	∂a	∂e		∂b	∂f		∂c	∂d	∂g	∂h

Figure 4-7: Illustration of PARAD performing reverse-mode AD on a parallel implementation of TwoByTwoMatVecSqLoss. This example continues the running example used in Figure 4-2 from Section 4.2 and Figure 4-5 from Section 4.3. The statement and operation stacks are presented in the style of the serial AD example in Figure 4-2, and the subarrays in these stacks that correspond to the subtapes D_1, D_2, and D_3 are indicated. Additional columns have been added to the statement and operation stacks to illustrate the S_{rcv}, O_S, and O_{snd} functions that map statements and operations to locations in the deposit array. For clarity, entries that would be undefined by these maps are marked as *input* or *output*. The deposit array state after processing each subtape during PARAD's right-first traversal over the tape is provided. The final gradients $\partial a, \partial b, \ldots, \partial h$ are obtained by summing the indicated subarrays of the deposit array.

processing a statement with the same gradient index. A single operation's contribution to the gradient, therefore, is visible to at most one statement.

We now examine the steps of PARAD.

The left-first traversal of the SPTape in Step 1 of Algorithm 1 maps each operation $O[i]$ to the last statement $S[j]$ that was encountered during the traversal for which $S[j].gid = O[i].gid$. The statement recorded in $O_s[i]$ is the statement that will extract $O[i]$'s contribution to the gradient.

In Step 2, first the array O^* is collected and semisorted to group together operations whose gradient contributions are extracted by the same statement. The step then processes subarrays $O^*[m..m+k]$ of operations in O^* and maps each operation to its index O_{snd} in O^*. At the same time, the step records in S_{rcv} the range of indices that will contain all gradient contributions that contribute to the value the statement associated with the group will extract.

Step 3 performs a right-first traversal of the SPTape. By Lemma 26, the serial right-first traversal of the SPTape processes statements and operations in the same order as SerialReversePass. When processing the statement $S[j]$ with statement index j we lookup in $S_{rcv}[j]$ the subarray in D containing all gradient contributions to $S[j].gid$ needed by statement $S[j]$, compute their sum, and set the subarray in D to zero. When processing the operation $O[i]$ the contribution of $O[i]$ to the gradient is deposited at location $D[O_{snd}[i]]$. The value extracted by statement $S[j]$ will exactly match the value extracted by the SerialReversePass algorithm when processing the same statement $S[j]$.

After the completing the right-first traversal over the tape, Step 4 accumulates the remaining gradient contributions deposited in D and places those contributions into the gradient table G. The two-phase semisort in Step 2 guarantees that all gradient contributions to the same gradient index are contiguous in D.

By construction, the serial execution of PARAD has identical behavior to the serial SerialReversePass procedure for performing reverse-mode AD.

To prove that PARAD is correct when run in parallel, it suffices to prove that the left-first and right-first traversals are data-race free. By construction PARAD has no races on the arrays $O, S, O_s, O_{snd}, S_{rcv}$. The remaining data accesses are to tables indexed by gradient indices. Since all accesses to these tables obey the safe adjoint access property, we can apply Lemma 27 to prove that the parallel traversals of the SPTape performed by PARAD are race-free. It follows that the gradient tables produced by PARAD and SerialReversePass are equivalent.

□

Analysis of PARAD

We now analyze the work and span of Algorithm 1.

Theorem 30 *Given a program $F : \mathbb{R}^n \to \mathbb{R}^m$ with total work $T_1(F)$ and span $T_\infty(F)$, the program R performing reverse-mode AD on F performs work $T_1(R) = \Theta(m \cdot T_1(F))$ with span $T_\infty(R) = \Theta(\log m + T_\infty(F) \log T_1(F))$.*

Proof. We analyze Algorithm 1 for $m = 1$. The bounds for larger m follow from spawning m instances of PARAD to process the m dimensions in parallel.

Step 1 performs a left-first traversal of the SPTape where $O(1)$ work is performed for each recorded statement and operation in the program F. We apply Lemma 28 with

$\sigma = O(1), \upsilon = O(1)$ to conclude that the total work and span of this step is $O(T_1(F))$ and $O(T_\infty(F))$ respectively.

Step 2 performs a parallel compaction and a parallel scan over operations in F, both of which perform $O(T_1(F))$ work and have $O(\log(T_1(F)))$ span. The semisort performed in this step can be performed in $O(T_1(F))$ work and $O(\log(T_1(F)))$ span by using the logarithmic-depth semisort from [140], which semisorts N elements in $\Theta(N)$ work and $\Theta(\log N)$ span.

Step 3 performs a right-first traversal with constant work per-operation. The work performed for each statement may be $\Omega(1)$, but each unit of work is associated with a unique operation. The amortized work for each operation and statement is, therefore, $O(1)$. Accumulating the subarray $D[m..m+k]$ requires $\Theta(\log k)$ span, and the worst-case span is $O(\log T_1(F))$. Therefore, we apply Lemma 28 with $\sigma = O(1)$ and $\upsilon = O(\log T_1(F))$ to conclude that this step performs $O(T_1(F))$ work and has $O(T_\infty(F) \log T_1(F))$ span.

Step 4 performs a parallel scan and multiple in-parallel reductions with total work $O(T_1(F))$ and span $O(\log T_1(F))$.

Thus, the work and span is bounded by the time required to perform the right-first traversal of the SPTape, resulting in total work $O(T_1(F))$ and span $(T_\infty(F) \log T_1(F))$. $\square$

4.5 Implementation of LibPARAD

This section describes LibPARAD which is an extension of the serial AD library Adept [156] that implements four different parallel algorithms for performing reverse-mode automatic differentiation, all of which employ the SPTape. Adept is a C++ library that implements reverse-mode AD via operator overloading. Other AD libraries using a similar approach include ADOL-C [340], Autograd [239], and PyTorch [280]. We chose Adept to base our implementation since its clever use of C++ expression templates and particularly concise representation for its tape data structure allows it to outperform other C++ AD libraries [325].

Implementation of the SPTape

LibPARAD uses a version of CSI that instruments the Tapir compiler representation of recursive fork-join parallelism [305] to insert operations in Figure 4-4 around **spawn** and **sync** statements, at the locations in the program code described in the figure.

LibPARAD modifies the parts of Adept that access its statement and operation stacks to instead use the SPTape data structure described in Section 4.3. We implemented the SPTape using a reducer hyperobject [115] and worker-local storage that is accessed by using the worker identifier returned by Cilk's GetWorkerNumber runtime call. Storage in data nodes for subtapes is allocated out of worker-local storage to improve efficiency in practice. The operations described in Figure 4-4 for constructing an SPTape are inserted into the program automatically at compile-time through the use of the CSI framework [302]. Specifically, LibPARAD uses a version of CSI that instruments the Tapir compiler representation of recursive fork-join parallelism to insert operations around **spawn** and **sync** statements. The LibPARAD library is designed to operate on any existing code using Adept, and does not require any additional annotations from the user aside from the normal use of Cilk keywords to express parallelism.

Our implementation deviates from the description in Figure 4-4 only slightly to handle nonbinary spawns. First, immediately after a **sync**, multiple PopShadow calls are performed to pop the S- and P-nodes pushed onto the shadow stack for each continuation of an executed **spawn** statement that the **sync** statement syncs. Second, the SPTape reducer hyperobject optimizes the handling chains of continuations, rather than simply enqueue calls to Combine to operate only on SPTape structures that capture the complete execution of a series-parallel subdag. Neither of these changes impact the theoretical work/span of the computation or the structure of the SPTape.

Implementation of PARAD

The implementation of PARAD includes optimizations related to the construction of the deposit array. In our discussions of Algorithm 1 in Section 4.4 we explained how to resolve write-write races by assigning operations unique memory locations from the deposit array to deposit their gradient contributions. Our implementation of PARAD avoids performing this work for operations that cannot possibly participate in a write-write race. We identify operations whose gradient index never appears in a statement and for such operations we accumulate gradients using worker-local sparse arrays. Additionally, we identify operations in subtapes whose contribution to the gradient is extracted by a statement in the same subtape. For such operations, we set its deposit location in D to be the global gradient table G. This does not introduce data races because it obeys the safe adjoint access property discussed in Section 4.3. The remaining operations are filtered and packed in-parallel and are processed as described in Algorithm 1.

Implementation of PARAD+S

A commonly used strategy to resolve races in reverse-mode AD is to employ worker-local (or thread-local) gradient tables. The problem with this approach, however, is that it is not work-efficient — extracting the gradient value for a statement requires work proportional to the number of processors, which may be greater than the number of operations that provided gradient contributions.

In PARAD+S, a sampling-based algorithm is used to identify "heavy" statements that accumulate a large number (greater than P) of operations. Operations contributing to "heavy" statements can use worker-local gradient tables instead of the deposit array without compromising the work-efficiency of the algorithm. As such, work can be avoided in Step 2 of Algorithm 1 by filtering these operations during the traversal of the SPTape to pack operations into O^*.

The additional computation performed by PARAD+S introduces a small constant overhead relative to PARAD on some benchmarks, but does not compromise the theoretical work-efficiency or scalability of PARAD.

4.6 Performance Evaluation

This section evaluates the performance of PARAD and PARAD+S that were implemented in LibPARAD. All experiments were run on an 18-core (hyperthreading disabled) Intel Xeon CPU (E5-2666 v3, 2.9GHz) with 64GB RAM available as a 4th-generation compute-optimized machine from Amazon web services. LibPARAD uses Cilk Plus to express parallelism and compiles the codes using the Tapir [305] based on LLVM 6.0.

Locks and Worker-Local implementations. We implemented two additional reverse-mode AD algorithms Locks and Worker-Local that employ fine-grain locking and worker-local storage, respectively, to resolve data races on the gradient table. Both of these implementations use the SPTape to record series-parallel dependencies and automatically parallelize the reverse pass over the tape, but they differ from LibPARAD in that do not need to perform additional work to organize a deposit array. This, however, comes at the expense of a loss of scalability and work-efficiency.

Application benchmarks. We evaluated the performance of LibPARAD across 8 benchmarks with different performance characteristics relevant to reverse-mode AD. For each benchmark, we measured the time required to perform AD while training the weights of the network via gradient descent. Figure 4-8 presents our performance results, including runtimes and speedup achieved for each benchmark. Figure 4-9 provides a summary of the performance achieved by each implementation (in geometric mean) over all 8 benchmarks.

Note on serial performance of Adept. Although, in general, Adept is highly efficient, we have observed on certain benchmarks (*mlp*, *gcn*, *lstm*) that the serial T_1 runtime of some of our algorithms outperforms the serial runtime T_s of Adept. This difference comes *entirely* from the forward pass of the computation, and we believe it relates to differences in how gradient identifiers are allocated in an SPTape and in Adept's serial data structures. This phenomenon also affects the algorithms using the SPTape, but negatively, on the *cnn* benchmarks, where it causes added overheads in the forward pass.

Multilayer perceptron benchmarks

The *mlp* benchmarks are feed-forward multilayer perceptron networks where *mlp1* has a single hidden layer of 800 nodes, and *mlp2* has two hidden layers with 400 and 100 nodes respectively. Both networks are trained on the MNIST data set [89]. Performance results are shown in Figure 4-8.

There is little performance variation among the different AD algorithms on the multilayer perception benchmarks. On these benchmarks, most of the work is performed by library calls (using OpenBLAS) to perform matrix-vector multiplication, and all implementations include an optimization of the Adept library for concisely recording the derivative dependencies resulting from a matrix-vector multiplication. Furthermore, the network is very simple and regular.

Both the Locks and the Worker-Local implementations achieve relatively good scalability on the multilayer perceptron benchmarks. On these benchmarks, the shared weight matrices are large and used only once-per-element of a training batch in the forward pass. The average number of operations-per-statement is large (> 100) which causes Worker-Local gradient tables to be relatively efficient on 18-cores. Since these operations' contributions are well distributed and are performed on large weight matrices, however, there is little lock contention and Locks also achieves good scalability on 18-cores.

Similarly, PARAD and PARAD+S perform similarly and achieve reasonably good scalability. The optimizations outlined in Section 4.5 for PARAD eliminate almost all the overheads related to the organization of the deposit array. As such, the 18-core runtime is slower, but only slightly, than Worker-Local, and is still better than Locks.

Convolutional neural networks

The *cnn1* and *cnn2* benchmarks are convolutional neural networks (CNNs) based on the *lenet-5* architectures. The *cnn1* benchmark implements a modernized version of *lenet-5*

Benchmark	Algorithm	T_s	T_1	T_{18}	T_s/T_1	T_s/T_{18}	T_1/T_{18}
mlp1	PARAD	149.50	156.94	14.44	0.95	10.35	10.87
mlp1	PARAD+S	149.50	165.18	15.17	0.91	9.85	10.89
mlp1	Locks	149.50	219.33	16.64	0.68	8.98	**13.18**
mlp1	Worker-Local	149.50	**129.87**	**12.91**	**1.15**	**11.58**	10.06
mlp2	PARAD	83.66	92.18	8.22	0.91	10.18	11.22
mlp2	PARAD+S	83.66	95.93	8.77	0.87	9.54	10.94
mlp2	Locks	83.66	95.75	8.31	0.87	10.07	**11.52**
mlp2	Worker-Local	83.66	**71.05**	**7.23**	**1.18**	**11.58**	9.83
gcn1	PARAD	94.30	113.00	13.30	0.83	7.09	8.50
gcn1	PARAD+S	94.30	111.00	**13.00**	0.85	**7.25**	**8.54**
gcn1	Locks	94.30	216.00	77.10	0.44	1.22	2.80
gcn1	Worker-Local	94.30	**84.90**	14.20	**1.11**	6.64	5.98
gcn2	PARAD	19.20	26.90	**4.14**	0.71	**4.64**	**6.50**
gcn2	PARAD+S	19.20	28.20	4.76	0.68	4.03	5.92
gcn2	Locks	19.20	47.40	9.38	0.41	2.05	5.05
gcn2	Worker-Local	19.20	**17.60**	5.63	**1.09**	3.41	3.13
cnn1	PARAD	126.00	307.00	**27.00**	0.41	**4.67**	**11.37**
cnn1	PARAD+S	126.00	311.00	28.60	0.41	4.41	10.87
cnn1	Locks	126.00	314.00	42.30	0.40	2.98	7.42
cnn1	Worker-Local	126.00	**246.00**	80.50	**0.51**	1.57	3.06
cnn2	PARAD	159.00	381.00	**35.30**	0.42	**4.50**	**10.79**
cnn2	PARAD+S	159.00	380.00	36.40	0.42	4.37	10.44
cnn2	Locks	159.00	566.00	197.00	0.28	0.81	2.87
cnn2	Worker-Local	159.00	**284.00**	94.10	**0.56**	1.69	3.02
lstm1	PARAD	168.00	249.00	46.60	0.67	3.61	5.34
lstm1	PARAD+S	168.00	248.00	44.80	0.68	3.75	5.54
lstm1	Locks	168.00	441.00	62.60	0.38	2.68	**7.04**
lstm1	Worker-Local	168.00	**147.00**	**34.20**	**1.14**	**4.91**	4.30
lstm2	PARAD	168.00	933.00	93.80	0.18	1.79	9.95
lstm2	PARAD+S	168.00	241.00	23.40	0.70	7.18	**10.30**
lstm2	Locks	168.00	442.00	71.00	0.38	2.37	6.23
lstm2	Worker-Local	168.00	**147.00**	**18.00**	**1.14**	**9.33**	8.17

Figure 4-8: Table of benchmark results for the 8 application benchmarks and 4 algorithm implementations. The best runtime/speedup on each benchmark is in bold font.

Algorithm	T_s/T_1	T_s/T_{18}	T_1/T_{18}
PARAD	0.57	5.12	9.02
PARAD+S	0.66	5.88	8.89
Locks	0.45	2.77	6.14
Worker-Local	0.94	4.96	5.27

Figure 4-9: Average (geometric mean) performance of algorithms over the 8 benchmarks.

using maxpooling layers and linear rectifiers as activation functions. The *cnn2* benchmark implements the *lenet-5* architecture as it was originally described in [210, 89] with average pooling and the use of the *tanh* activation function. For both networks, we verified that we achieve the expected accuracy for these well known networks after training for sufficiently many epochs. The performance results are shown in Figure 4-8.

The Worker-Local and Locks implementations scale poorly on the *cnn* benchmarks. The average number of operations-per-statement is approximately 2 on *cnn1* and 8 on *cnn2*. As such, Worker-Local performs substantially more work when executing on 18-cores than it does when it executes serially. As such, Worker-Local achieves just 3x self-relative speedup on the *cnn* benchmarks on 18-cores. Locks performs similarly poorly on the *cnn2* benchmark, but performs better on *cnn1* — achieving 7.4x self-relative speedup. Both of the *cnn* benchmarks have many parallel updates to small shared weight matrices which causes scalability issues for Locks. The *cnn1* benchmark, however, has a lot of dynamic sparsity (many gradients are zero) which substantially reduces lock contention.

The PARAD and PARAD+S algorithms scale well on the *cnn* benchmarks each achieving 10–11x self-relative speedup. Differences between the SPTape and Adept stack data structures results in a performance hit on these benchmarks that causes somewhat worse overheads. As a result, the speedup relative to Adept on *cnn* is about 4.5x on 18-cores for PARAD and PARAD+S.

Graph convolutional networks

The graph convolution network (GCN) benchmark *gcn1* performs community detection on the *pubmed* network [195]. The input embeddings for the vertices are sparse bags of words vectors and the network uses a single graph convolutional layer that learns an embedding of dimension 32 for each vertex in the graph. We follow the training method described in [69] and match their accuracy when training for the same number of epochs. The *gcn2* benchmark operates on the *email-Eu-core* network dataset [221, 350] with random feature vectors of dimension 1024 and learns an embedding of dimension 64. The performance results are provided in Figure 4-8.

The scalability of Locks and Worker-Local is mixed on the *gcn* benchmarks. Like *cnn* there are many parallel updates to small weight matrices, but the degree of contention varies since it depends on the structure of the graph. As such, the self-relative speedup of Locks and Worker-Local is mixed. The speedup relative to the serial Adept implementation, however, is uniformly poor (1.2–2x) for Locks on 18-cores. Worker-Local achieves 6.6x speedup relative to Adept on *gcn1* and 3.4x speedup on *gcn2*.

The scalability of PARAD and PARAD+S is better than Locks and Worker-Local on both *gcn* benchmarks. On *gcn1* they achieve 8.5x self-relative speedup, and on *gcn2* achieve 6–6.5x self-relative speedup. Relative to the serial adept code, they are about 7x faster on *gcn1* and 4–4.6x faster on *gcn2* on 18-cores. The PARAD+S algorithm is the best performing algorithm on *gcn1*, by a small margin. The more complex parallel structure in the *gcn* benchmarks causes the more sophisticated optimizations in PARAD+S to be, subtly, visible.

Long short-term memory networks

The long short-term memory network (LSTM) [155] benchmarks *lstm1* and *lstm2* implement a recurrent neural network to generate text given examples. We used a subset of the Paul Graham dataset consisting of 500 100-character data points, using a one-hot encoding of each character. The primary difference between *lstm1* and *lstm2* is that *lstm1* has very little parallelism. The *lstm2* benchmark expresses additional fine-grained parallelism and allows some algorithms to achieve improved performance.

On the *lstm* benchmarks, the Worker-Local implementation performs very well and Locks performs poorly. The locking overheads are significant on the *lstm* networks and so the scalability of Locks relative to Adept is poor 2x, even though its self-relative speedup is fairly good. The Worker-Local implementation scales relatively well achieving 4x and 8x self-relative speedup on *lstm1* and *lstm2* respectively. Worker-Local is pretty efficient relative to Adept on the *lstm* benchmarks as well since there are hundreds of operations, on average, per statement. As such, Worker-Local achieves 5x and 9x speedup relative to Adept on *lstm1* and *lstm2*.

The differences between PARAD and PARAD+S are very apparent in the *lstm* benchmarks. Although they perform similarly to one another on *lstm1*, the expression of additional fine-grained parallelism in *lstm2* causes an increase in PARAD's constant overheads. The additional optimizations in PARAD+S, however, enable it to perform consistently across both *lstm1* and *lstm2*. PARAD+S achieves 7x and 10.3x self-relative speedup on *lstm1* and *lstm2* respectively, and achieves 3.7x and 7x speedup relative to the serial Adept code. Although not as good as Worker-Local on these benchmarks, PARAD+S is not too far behind in terms of performance and scalability.

4.7 Related work

This section discusses related work on parallel AD.

Some previous work has examined parallelization of forward-mode AD. Forward-mode differentiation can be accomplished using ***dual numbers*** that tie to each parameter x the infinitesimal ϵ_x and perform computations on $\tilde{x} = (x + \epsilon_x)$. Hovland *et al.* have explored approaches that augment MPI communications to transmit dual numbers between nodes, in order to parallelizing forward-mode AD for MPI programs [159, 158]. Forward-mode AD is less efficient than reverse-mode AD, however, for many applications, including machine-learning applications, for which the function $F : \mathbb{R}^n \to \mathbb{R}^m$ being differentiated has a low-dimensional output, that is, $m \ll n$. Bücker *et al.* [55] examine parallel forward and reverse-mode AD for OpenMP programs when $\min m, n$ is large. In contrast, PARAD's parallelization of reverse-mode AD is work-efficient and scalable, even when $m = 1$.

Prior work has developed specialized parallel reverse-mode AD algorithms for specific computations. Gremse *et al.* developed optimized reverse-mode AD computations on GPUs of four input functions [138]. Hückelheim *et al.* developed a parallel reverse-mode AD algorithm for compressible flow solvers for unstructured meshes [161]. Other parallel reverse-mode AD algorithms have been devised for stencil computations [164, 163], and convolutional neural networks [162]. The code transformations employed for these parallel reverse-mode AD algorithms do not generalize to handle arbitrary recursive fork-join programs. PARAD, meanwhile, handles arbitrary recursive fork-join parallel programs.

Prior work has developed several systems that support parallel reverse-mode AD. In the context of a parallel plasma simulation code, Bischof *et al.* explored parallel reverse-mode AD for OpenMP programs, using OpenMP-thread-local instances of ADOL-C [28]. Substantial prior work has explored parallel reverse-mode AD in message-passing programs [158, 159]. Schanen *et al.* [301] have developed an adjoint MPI library, which provides appropriate MPI communications to parallelize reverse-mode AD, based on communications in a given MPI program. These systems are not guaranteed to be work-efficient, because of the overheads they incur to combine parallel gradients. In contrast, the PARAD parallel AD algorithm targets shared-memory multicore systems and is provably work-efficient and scalable.

4.8 Conclusion

This chapter presented PARAD, a work-efficient and scalable reverse-mode AD algorithm for determinacy-race-free recursive fork-join programs. PARAD performs reverse-mode AD on a given program with scalability similar to the input program and bounded contention. We have observed that PARAD works well in practice, achieving good work efficiency and self-relative speedups on eight machine-learning benchmarks.

Chapter 5

A Multicore Path to Connectomics-on-Demand

This chapter presents, as a proof-of-concept, a high-throughput connectomics-on-demand system that runs on a multicore machine with less than 100 cores and extracts connectomes at the terabyte per hour pace of modern electron microscopes. This work was conducted in collaboration with Alexander Matveev, Yaron Meirovitch, Hayk Saribekyan, Wiktor Jakubiuk, Gergely Odor, David Budden, Aleksandar Zlateski, and Nir Shavit.

Abstract

The current design trend in large scale machine learning is to use distributed clusters of CPUs and GPUs with MapReduce-style programming. Some have been led to believe that this type of horizontal scaling can reduce or even eliminate the need for algorithm development, careful parallelization, and performance engineering. This chapter is a case study showing the contrary: that the benefits of algorithms, parallelization, and engineering, can sometimes be so vast that it is possible to solve "cluster-scale" problems on a single commodity multicore machine.

Connectomics is an emerging area of neurobiology that uses cutting edge machine learning and image processing to extract brain connectivity graphs from electron microscopy images. It has long been assumed that the processing of connectomics data will require mass storage, farms of CPU/GPUs, and will take months (if not years) of processing time. We present a high-throughput connectomics-on-demand system that runs on a multicore machine with less than 100 cores and extracts connectomes at the terabyte per hour pace of modern electron microscopes.

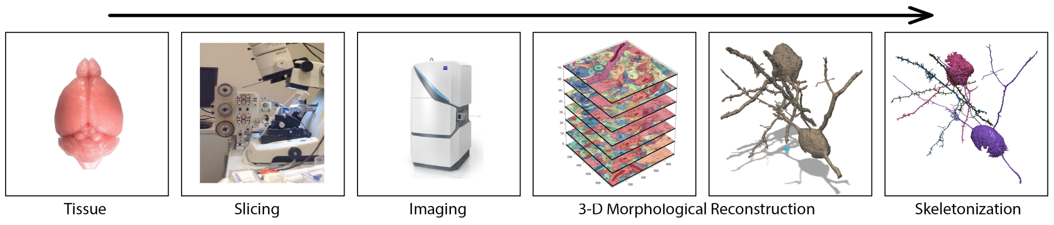


Figure 5-1: The Stages of a high-throughput connectomics pipeline.

5.1 Introduction

The conventional wisdom in machine learning is that large scale "big-data" problems should be addressed using distributed clusters of CPUs, GPUs or specialized tensor processing hardware in the cloud [71, 80, 224, 329, 147, 135, 166]. This chapter presents a solution to one of the most demanding big-data machine learning areas: connectomics. It provides a case study in how novel algorithms combined with proper parallelization and performance engineering, can reduce the problem from a large cluster to a single commodity multicore server (that can be placed in any neurobiology lab), eliminating the crucial bottleneck of transferring data to the cloud at terabyte-an-hour rates.

5.1.1 High-throughput connectomics

Perhaps neuroscience's greatest challenge is to provide a theory that accommodates the highly complicated structure and function of neural circuits. These circuits, found in flies, in mammals, and ultimately, in humans, are conjectured to be the substrate that enables the complex behavior we call "thought." However, to this day, our ability to provide such a theory has been hampered by our limited ability to view even small fragments of these circuits in their entirety. Neurobiologists have had sparse circuit maps of mammalian brains for over a century [289]. However, as surprising as this may seem, no one has seen the dense connectivity of even a single neuron in cortex, that is, all of its input and output connections (called synapses) to other neurons. No one has been able to map the full connectivity among even a small group of neighboring neurons, not to mention the tens of thousands of neurons that constitute a "cortical column." Without such maps, it would seem hard to understand how a brain consisting of billions of interconnected neurons actually works. Would you believe that someone understood how a modern microprocessor worked if they could tell you in great detail how individual transistors operated but were unable to describe how these transistors are interconnected?

Mapping brain networks at the level of synaptic connections, a field called *connectomics*, began in the 1970s with a study of the 302-neuron nervous system of a worm [343]. This study, which required capturing and combining hundreds of electron microscopy images, was done by hand and took 8 years. Manual mapping of minute volumes of neural tissue have recently produced breakthrough results in neuroscience [191, 217, 262]. However, extending the approach to the millions and billions of neurons in higher animals seems an unattainable goal without a fast and fully automated pipeline.

Recent advances in the design of multi-beam electron microscopes now allow researchers to collect the nanometer resolution images, necessary to view neurons and the synaptic connections between them, at unprecedented rates. A cubic millimeter of brain tissue, enough to contain a mouse cortical column, is within reach. This is only a tiny sliver of brain, about the size of a grain of sand, but it will contain about 100 thousand neurons and a billion synapses. The cubic millimeter will constitute about two petabytes of imagery that will be collected in about 6 months using a 61-beam electron microscope that generates half a terabyte of imagery per hour [100, 226].

A modern connectomics pipeline [227], as depicted in Figure 5-1, consists of a physical part and a computational part. The physical part takes a piece of stained brain tissue embedded in a resin, slices it thousands of times using a special microtome device, and feeds these minute slices into an electron microscope that scans them and produces separate images of the slices [100, 309]. The computational part of the pipeline, the focus of this

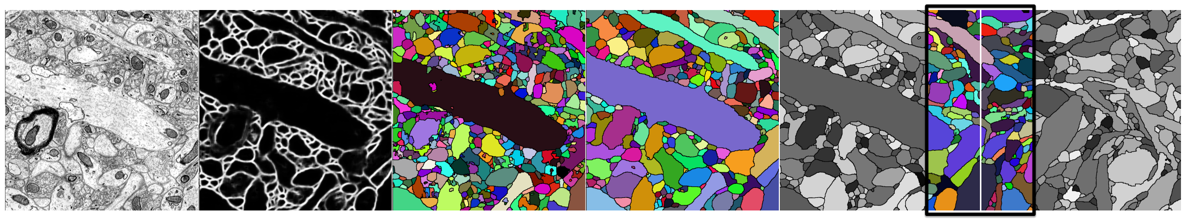


Figure 5-2: 2D visualization of connectomics pipeline stages: (a) Electron microscope (EM) image. (b) Membrane probabilities generated by CNN. (c) Over-segmentation generated by Watershed. (d) Neuron reconstruction generated by agglomeration. (e) Pipeline inter-block merging: for each two adjacent blocks, the pipeline slices a boundary sub-block and executes agglomeration.

chapter, then takes the thousands of separate 2-dimensional images, reconstructs the 3-dimensional neurons within them, and produces skeletonizations or graphs that capture their morphological and connectivity properties.

However, a viable solution to the computational problem of extracting the skeletonizations and connectivity maps from the image data still seems far away. Using existing algorithms and at the present computing rates, the common assumption is that extracting the connectivity of the circuits within two petabytes of data may take years and require supercomputing clusters [226, 338]. It is even unclear how to efficiently move the vast amounts of data from the microscope to a large scale storage and compute facility where it will be processed [227]. The prospect of connectomics-on-demand, where neurobiology labs around the world each run their own microscopes and extract connectomes "as needed" seems far far away.

This chapter takes a first step towards proving the feasibility of designing a high-throughput connectomics-on-demand system that runs on a multicore machine with less than 100 cores and extracts connectomes at the terabyte-per-hour pace of modern microscopes. Such a system, once achieved, will eliminate the need to transfer and store petabytes of data in special warehouses. It could be readily deployed in labs across the world, allowing scientists to view connectomes as they come off the scope. Down the road, such efficient connectomics systems could make it feasible for neurobiologists to extract the complete connectome of a mouse cortex of about 12 million neurons and 120 billion synapses from about 100 petabytes of data, and eventually make it possible to analyze exabyte-scale parts of the connectome of both healthy and diseased human brains (Human brain has roughly 100 billion neurons and a quadrillion synapses).[1]

5.1.2 Towards an automated terabyte-per-hour connectomics pipeline

The "proof of concept" connectomics system we present here processes a terabyte of data, from image-stack to detailed skeletons, in less than **4 hours** on a single 72 core Haswell-based multicore machine with 500GB of memory. The upcoming generation of both GPU

[1]These numbers may sound like science fiction, yet as Lichtman and Sanes note by analogy [228], sequencing the first human genome took multiple labs around the world a concerted effort over 15 years, while today a single lab can sequence a human genome within hours.

and CPU chips, with some further optimizations (see our performance section), easily place it within the target terabyte-an-hour performance envelope of todays fastest electron microscopes. One should contrast this with the recent connectomics system of Roncal *et al.* [294] that uses a cluster of 100 AMD Opteron cores, 1 terabyte of memory, and 27 GeForce GTX Titan cards to process a terabyte in **4.5 weeks**, and the state-of-the-art distributed MapReduce based system of Plaza and Berg [284] that uses 512 cores and 2.9 terabytes of memory to process a terabyte of data in **140 hours** (not including skeletonization). Importantly, the speed of our pipeline does not come at the expense of accuracy, which is on par or better than existing systems in the literature [294, 192, 284] (using the accepted *variation of information* (VI) measure [252]).

Our high-level pipeline design builds on prior work [192, 241, 271, 272, 278, 284]. It passes the data through several stages as seen in Figure 5-2. The image data is split into blocks. The pipeline first runs a convolutional neural network (CNN) on the separate image blocks to detect the boundaries of neurons. The neuronal objects defined by the boundaries are then segmented into objects by a watershed algorithm. At this point the image is over-segmented, that is, neuronal objects are fragmented. The next step in the reconstruction is to run an agglomeration phase that merges all the fragments in a block into complete objects. Then the blocks are merged, so that objects now span the entire volume. Finally, as depicted in Figure 5-8, the objects are skeletonized.

5.1.3 Our contributions

- A new CPU execution engine for CNNs that scales well on multicore systems, processing 1024x1024 images 70x faster than previous state-of-the-art [359] on a single 18-core Haswell CPU. Our code applies GCC-Cilk (a state-of-the-art work-stealing scheduler [37]), is optimized to leverage Intel AVX2 instructions [345] and maximizes memory reuse in L1, L2 and L3 cache sub-systems. Analysis shows that this code achieves 80% utilization of the peak theoretical FLOPS of the system on a single thread and scales almost linearly, compared to previous approaches (Caffe CNN framework with Intel MKL [177, 78]) that exhibit only 20% utilization and no scalability beyond 4 threads. The remaining performance gap is obtained by implementing minimal "dense" (fully-convolutional) CNN inference, which involves 284x fewer computations than naive implementations.

- A new GPU execution engine for CNNs that leverages custom fast Fourier transforms (FFTs) and the latest cuDNN primitives [70]. The system performs ∼2x faster for our CNN benchmarks than previous state-of-the-art [264] on a Titan X GPU.

- Parallelized and performance engineered version of NeuroProof, a system originally developed in Janelia Research Labs by Parag, Chakrobarty and Plaza [279]. This system implements an agglomeration procedure that reconstructs 3D neurons from the watershed oversegmentation. Specifically, we redesigned the Regional Adjacency Graph (RAG) algorithm as an augmented mergesort, efficiently defer expensive operations that occur during constructions and traversals of RAGs, and batch computations for lazy execution. Our implementation is 6.5x faster on a single thread, and has 85x scalability factor on our 72-core server (super-linear factor due to HyperThreading).

- Two new algorithms for the connectomics domain: (1) a new inter-block merging algorithm that, unlike prior approaches, applies parallelized NeuroProof to optimize object-pair merges, and (2) a parallel skeletonization algorithm that uses novel techniques and GCC-Cilk based chromatic scheduling [183] to execute efficiently on multicores. We describe the algorithmic side in more detail in [254].

We believe that our work departs from existing systems by showing that combination of new algorithms, proper parallelization of existing algorithms, modern scheduling using languages like Cilk, and performance engineering of code to fully utilize CPU resources, can move the connectomics problem from the large scale distributed systems space to run on a commodity shared-memory multicore system. This might be true for other big-data machine-learning based problems in medicine and the life sciences, and even for problems where big systems are necessary, our approach may provide insights on how to make individual system elements much faster and scalable.

5.1.4 Related work

The image segmentation approach we use was developed by [8] and suggested for automatic electron microscopy segmentation by [271]. It is used by most of todays systems [271, 272, 278, 241, 192, 284]. Increasingly accurate membrane predictors based on convolutional neural networks have recently been proposed [73, 295]. Our system implements a solution that is inspired by these algorithms, is on-par with their current accuracy levels, and yet is simpler and faster to execute. Other studies have focused on new approaches to segmentation of supervoxels (e.g., [241]). Further research however is required to benchmark their accuracy on even gigabyte size datasets, so at this time their scalability cannot be addressed.

We are aware of three previous connectomics pipelines: the original RhoAna pipeline by Kaynig *et al.* [192], the system by Roncal *et al.* that extends RhoAna [294], and state-of-the-art Plaza and Berg's Spark-based system [284] that uses NeuroProof.

5.2 System overview

This is the story of how one can replace the use of a large distributed system with a single multicore machine. Although a large distributed system may have tremendous computational resources at its disposal, its raw computing power comes with a cost of additional overheads and system complexity, e.g. data must be moved over the network, and the system must support a degree of fault tolerance [284, 71, 356, 225, 357, 86]. These overheads tend to be small for problems that are embarrassingly parallel, but can come to dominate the execution time of more complex computations that need to operate on shared data. In addition, the opportunities to obtain performance within a single multicore are abundant, and can result in performance improvements that rival or even dwarf those obtained through horizontal scaling. As we will see, the connectomics problem requirements can be satisfied using a single multicore machine to execute a pipeline of carefully designed multicore algorithms that efficiently utilize a machine's available computing cycles, take advantage of the low shared-memory communication overheads, and parallelize across cores using efficient task scheduling tools.

5.2.1 Pipeline structure overview

The input to our pipeline is a sequence of aligned 2D images each of which represents a single slice of a 3D volume of brain that was captured at high resolution ($\sim$3-4nm) by an electron microscope. These images are broken down into smaller sub-images with a standardized size of 1024x1024 pixels, which are then grouped into blocks of 1024x1024x100 pixels. The pipeline executes a segmentation procedure that extracts the neuronal objects within each block, and then executes an inter-block merging procedure that "combines" the per-block segmentations to obtain a complete segmentation of the original input volume [254]. These stages of the connectomics pipeline are illustrated in 2D within Figure 5-2.

First, as seen in Parts 1 and 2 of Figure 5-2, a convolutional neural network (CNN) is executed to detect membranes (or cell borders). This network was trained using "ground truth" annotation by human experts. The training process is time-consuming, but since it only needs to be performed once on an annotated subset of the dataset it does not impact the pipeline's throughput. The result of applying the CNN to a subvolume is a new 1024x1024x100 block where each pixel indicates a membrane probability between 0 and 1. Since the CNN may run on blocks independently, this stage of the pipeline may be executed in a distributed manner. It turns out, however, that careful performance engineering enabled this stage of the pipeline to execute sufficiently fast on a single multicore machine. In fact, we found that our optimized code for executing the CNN was able to significantly outperform existing state-of-the-art CPU [177, 78, 359] and GPU [264] implementations. In Section 5.3 we describe the design of this CNN, and the steps taken to ensure that the our implementations made efficient use of the machine's compute cycles, L1/L2/L3 caches, and disk.

Next, as seen in Part 3 of Figure 5-2, a 3D watershed algorithm executes on the membrane probability map. The watershed algorithm performs a BFS-style flood from "seed" pixels that have 0 probability of being a membrane, to pixels with higher probability. The result of the watershed execution is a new block in which segments (3D objects) have been formed around seeds, each with a segment identifier. Methods of parallelizing watershed have been described in the literature (e.g. [261]), but it turned out that our pipeline was able to achieve better performance by engineering an efficient sequential algorithm with low-memory consumption. This strategy allowed us to obtain a watershed algorithm that is an order of magnitude faster than the code provided in the popular OpenCV [174] library, and whose low-memory requirements permit us to run many independent instances of the algorithm on a shared memory machine. Section 5.4 describes the design of this serial watershed algorithm in greater detail.

The segments produced by the watershed algorithm represent an over-segmentation of the true neuronal objects; i.e. each neuron might be fragmented into many smaller parts, as shown in Part 3 of Figure 5-2. To obtain segments that represent whole neurons an agglomeration algorithm is employed that merges segments that lie within the same 3D object. The result of the agglomeration procedure is a collection of larger segments that each represent a whole neuronal object, as is illustrated (in 2D) in Part 4 of Figure 5-2. The agglomeration procedure used by the pipeline is based upon the serial NeuroProof agglomeration algorithm of Parag *et al.* [279]. We reformulated the Neuroproof procedure to use parallel algorithms, and performed a variety of performance optimizations to reduce the total work and memory usage. Together, these optimizations obtained a 6.5x speedup on 4-cores over the original sequential code. Moreover, we achieve near-linear scalability which provides an additional 82x factor speedup on our 72-core server with HyperThreading. Section 5.5 describes, in greater detail, the parallelization techniques and performance optimizations that were employed to optimize the agglomeration stage of the pipeline.

After a segmentation has been obtained for each block, the pipeline executes an efficient inter-block merging procedure on the reconstructed blocks [254]. Here we utilize the shared memory properties of the machine in which I/O operations are automatically cached by the OS kernel. Thus, for every two adjacent blocks, our block merging algorithm carves out a thin sub-block near the shared boundary and employs a variant of our parallel agglomeration procedure to identify and merge objects across this boundary. Part 5 of Figure 5-2 is a two dimensional rendering of this merging process. The result of each merging is a small file, and because all the files are on the same machine, it is inexpensive to combine all of them into a full segmentation. This contrasts sharply with the approach of Plaza and Berg [284] where a sophisticated merge algorithm must be executed across many machines in the network. In our case, there is no need to perform expensive data-transfers over the network and to support a complex failure detection and recovery mechanisms, since the whole system executes on a single server. The design of the inter-block merging procedure is discussed further in Section 5.6.

The final step of the pipeline skeletonizes the volume segmentation (see Figure 5-8) on a per-block basis. A skeleton provides a space-efficient one dimensional representation of a 3D volume that runs a long the volume's medial axis, allowing for faster and easier analysis of the biological structures. It is also an intermediate step on the way to creating a graph representation of the neuronal objects. We experimented with various skeletonization algorithms and found out that a subfield "thinning" algorithm [25] fits our purposes best. with points on the object boundary and repeatedly removes ones that do not affect overall topological connectivity. We devised a simple and efficient parallel algorithm for extracting volume skeletons using chromatic scheduling [183] to efficiently schedule the parallel order of which points are considered for deletion doing the thinning process. The details of the skeletonization algorithm are discussed further in Section 5.7.

5.3 Segmentation with CNNs

Our connectomics pipeline leverages a CNN to map EM input images to membrane probability output images. A CNN system is typically made up of two components: the *network architecture* that defines layers of perceptrons and their connectivity, and the *computational framework* that executes this architecture's forward propagation over a trained network. typically be trained using backward propagation and ground truth data, a compute intensive phase that needs to be performed once. The trained network is then executed multiple times in the life of the connectomics pipeline, allowing us to focus on forward propagation times.

In this section we describe our network architecture and new computational frameworks for both CPU and GPU. Specifically, these frameworks implement fully-convolutional neural networks by applying "dense computation" rather than a patch-based sliding window approach [73]. This is accomplished through the usage of "max-pooling fragments," which apply the traditional maxpooling operation to different offsets of the input image [128, 245]. The resultant images are then recombined to produce the final dense result.

In contrast to other methods of dense computation (*e.g.,* dilated convolution [352], strided kernels [332], max filtering [215, 359] and filter rarefaction [231]), we adopt maxpool fragments because they generate independent matrices that are contiguous in memory. This feature is leveraged for improved parallelization of both our CPU and GPU-based implementations, as described in detail in Sections 5.3.2 and 5.3.3 respectively.

5.3.1 Our network architecture

We recently proposed a CNN architecture, called *MaxoutNet* [254, 196, 191], which consists of 4 *ConvPool* layers of alternating convolution/maxpool pairs aggregated with a maxout function. Convolutional layers use 4×4 kernels with stride 1 and either 8 or 32 channels, which combined with stride 2 max-pooling yields a 105×105 field-of-view for each output pixel. In our pipeline execution, MaxoutNet leverages the fact that our input EM images are 3nm high resolution (2048×2048): it performs sub-sampling of the input by executing standard pooling in the first ConvPool layer, while executing "dense" poolings in the next layers. This produces a 2-fold subsampled 1024×1024 output image, and accelerates the network by a factor of 4 without losing too much accuracy. For our benchmarks, described in Sections 5.3.2 and 5.3.3, we simplify this network and execute only the last 3 layers of *ConvPool*.

5.3.2 A fast CPU framework for CNNs

We introduce *XNN*, a CNN framework implemented in C and inline assembly for Haswell CPUs. These provide support for AVX2 instructions and include two FMA units, allowing the chip to execute 32 floating-point operations per cycle [345]. As a result, a single 2.5 GHz 18-core Haswell chip can theoretically produce 1.44 TFLOPS (compared to state-of-the-art GPU, like NVIDIA's Titan X, that theoretically produces 6TFLOPS). In practice it is challenging to achieve close to these peak floating-point utilizations, requiring careful engineering techniques that take into account the hardware specifics. The remainder of this section describes the techniques we applied to achieve state-of-the-art CNN throughput.

Haswell AVX2 SIMD parallelization

We initially adopted Intel's MKL convolution primitives [78]. However, our results showed that the CPU FLOPS utilization for MKL was only $\sim$20%, so we reimplemented this critical component with hand-crafted AVX2 assembly. One of complication of AVX2 vector instructions is that they must have aligned memory addresses. As a result, it is impossible to slide a convolution window on a 2D matrix by using AVX instructions since some of the window locations are not aligned. Instead, we leveraged the fact that CNN's internal matrices are 3D, where the first two dimensions are spatial and the third dimension is channels. The convolution window slides on the first two spatial dimensions while the AVX applies to the third channel dimension, where the number of channels is a multiple of the AVX vector width. This constraint is not problematic for CNNs that already use many channels, thus allowing us to achieves 70-80% utilization.

Our first implementation was a simple loop over the channel dimension, which did not provide a high CPU floating-point utilization. To improve this, we computed 6 convolutions at the same time by using a different set of AVX registers for each (a total of 16 registers). We interleave the register load, FMA compute, and store operations of the 6 running convolutions in a way that maximizes the usage of the two FMA units of the Haswell CPUs. In addition, we made the loop bounds and counter increments constant at compile-time, so that the GCC vector optimizations could unroll the loops and optimize the code as much as possible.

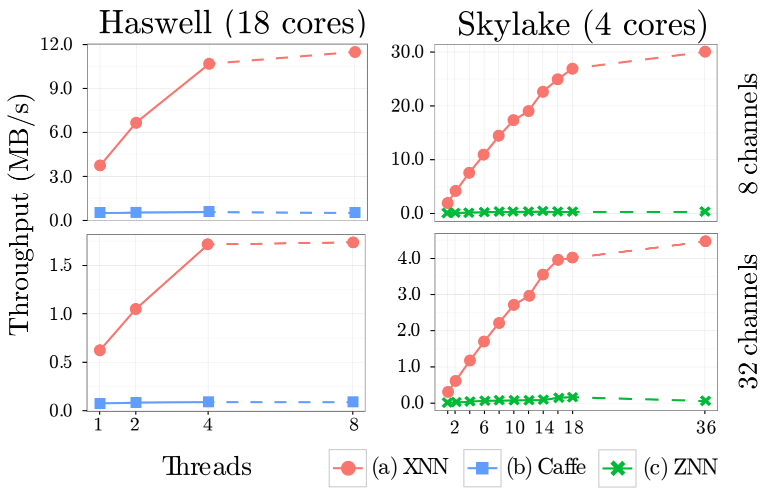


Figure 5-3: Throughput of the three CPU-based CNN implementations evaluated using 8 and 32-channel MaxoutNet.

Cilk-based concurrency and caching

We used Cilk [115, 219], a work-stealing scheduler that is supported by GCC 4.8, to dynamically generate multicore fine-grained tasks. The key advantage of Cilk is that it provides a "fork-join" primitive that scales well to many cores. It uses a sophisticated combination of per-thread double-ended queues, and a clever work-stealing algorithm that has no bottlenecks and memory contention on a shared memory system.

Our implementation applies the Cilk "fork-join" primitive to all "for-loops" that have no loop iteration dependencies, simplifying our parallelization effort. However, we still require that memory accessed by executing Cilk threads fits into the L3 cache. The L3 cache is much faster than main memory, and having threads working on data that resides in L3 is key to ensuring a high ratio of compute to memory access, which is important for scalability. For example, large matrix operations (*e.g.* convolution) are executed over sub-matrices that each fit into the L3 cache, with a set of threads operating over this sub-matrix before proceeding to the next. We observed that this approach improves scalability by a factor of 3-4x.

Memory usage

Another important aspect of the implementation is the amount of memory used to compute the forward propagation pass of the CNN network. Since we are only concerned with the performance of forward propagation, we optimized the memory usage significantly by allocating only two large matrix buffers, one for input and one for output, and then reuse (swap) these buffers between layers. As a result, the memory usage is bounded by the largest input/output size of a single CNN layer, which reduces cache trashing and OS paging effects. Note that memory is only allocated at the start, *i.e.* there are no dynamic memory allocations during the computation itself that could introduce contention and bottlenecks.

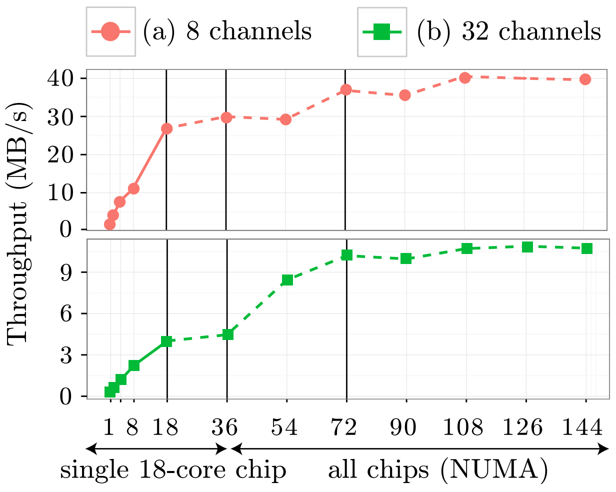


Figure 5-4: Throughput of the XNN up to 144 hardware threads on our 4 socket system.

CPU benchmarking

Benchmarking was performed on the same shared memory server we use to run our pipeline: a 4-socket machine with Intel *Haswell* 18 core chips and 512 GB of RAM. We also benchmarked a machine equipped with a single 4-core Intel *Skylake* CPU with 64 GB RAM.

We benchmarked our XNN framework against the most efficient known multicore CNN frameworks the Caffe framework of Yangqing *et al.* [177] and the ZNN framework of Zlateski *et al.* [359]. Throughput was evaluated using 1024×1024 (6 nm) images. We benchmarked all three using a lightweight version of MaxoutNet with 3 fully convolutional ConvPool layers (field-of-view = 53) because the subsampling required for the fourth layer (3 nm data) is available only in our XNN framework. We benchmark for both 8 and 32 channels. To ensure best performance, we compiled with the Intel C++ Compiler (ICC) version 16.0.0, optimized for maximum speed (`-O3`), linked against Intel Threading Building Blocks (TBB) version 4.4 (`libtbbmalloc`, `libtbbmalloc_proxy`), and Intel Math Kernel Library (MKL) version 11.3 with enabled FFT caching, and single-precision floating-point arithmetic.

The results of this benchmarking are presented in Figure 5-3. In this execution, we perform a scalability test on a single 18-core chip: an input is set to a fixed size and we increase the number of threads. By binding multi-threading to one chip we disable the NUMA-effects for this test (expensive inter-chip communication). It is evident that our XNN CPU framework exhibits substantial throughput improvement over Caffe and ZNN. First, one can see that there is a significant difference in single-threaded performance, which is the effect of the hand-crafted assembly of XNN that utilizes the two FMA AVX2 units of Haswell CPU. Second, the scalability of XNN is almost linear, 15x over a single thread on 18 cores. This is achieved by constraining active threads to operate on memory sets that fit into the L3 cache while using Cilk to manage short-living jobs.

Figure 5-4 presents the results of XNN executed as a single instance (fixed input) on all 72 cores across 4 sockets, running up to 144 threads to utilize hyperthreading. As can be seen, for the 32 channel execution (green) linear scalability is evident up to 18 cores and plateaus all the way to 36 cores, as hyperthreading provides only 10-20% boost in performance. Increasing threads from 36 to 72 utilizes additional cores to again provide pseudo-linear scalability. Beyond 72 cores, additional threads leverage hyperthreading for

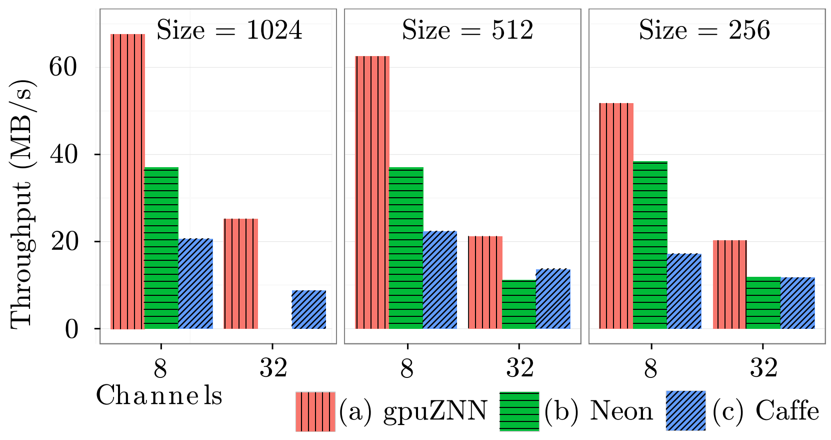


Figure 5-5: Throughput of GPU-based CNNs using 8 and 32-channel MaxoutNet architectures.

Method	**Type**	**8-channel MaxoutNet Throughput (MB/s)**	**32-channel MaxoutNet Throughput (MB/s)**
XNN	CPU (72-core)	111.1	16.67
gpuZNN	GPU (Titan X)	67.61	25.28
Neon	GPU (Titan X)	37.06	(exceeds memory)
XNN	CPU (4-core)	11.49	1.74

Figure 5-6: A comparison of CNN throughput for the best-performing CPU and GPU-based implementations using the MaxoutNet architecture.

a minimal further improvement. For 8 channels (red), the smaller network size generates low latency Cilk tasks that introduce more pressure on the software implementation. There is still linear scalability up to 18 threads, but only small improvement is evident beyond 36 cores. This is because the overhead of threading (NUMA side effects) is substantial compared to the performance benefit from adding additional cores.

In our pipeline execution, we overcome these NUMA overheads by combining multi-processing with multi-threading. Specifically, we execute multiple instances of XNN (each bound to a specific chip) with an empirically optimized number of threads, resulting in a 2.8x speedup. Generating these instances increases the RAM requirements (not problematic for our 512 GB RAM system).

5.3.3 A fast GPU framework for CNNs

In this section we present *gpuZNN*, a GPU-based CNN framework that improves upon the previously published ZNN [359] to optimize forward propagation throughput.

The SIMT programming model and limited amount of memory available on a GPU require a different implementation than our XNN CPU framework. To saturate the GPU, we process multiple input images simultaneously. We also leverage the fact that all sub-samplings of the "dense" computation have equal size, which is equivalent to processing multiple inputs of a layer at the same time.

Most of our convolutional primitives use cuDNN's low-level implementations [273, 70]. To address the large memory overhead of some primitives, we allow the computation to be split into multiple stages, computing a subset of results at the time. This is accomplished by computing a subset of input batches and/or subset of output images (feature maps). This can reduce the parallelization potential, but reducing the memory overhead can allow for using a computationally cheaper primitive. We also implement optimized FFT-based convolutional primitives with very low memory overhead. Batched 1D FFTs and memory reshuffling was used to efficiently utilize the GPU.

Sub-sampling layers (maxpooling and maxout) were implemented using cuDNN's max-pooling primitive. We minimized the number of calls to the primitives such that each primitive performs more computation and can thus saturate the GPU. To allow this, we implicitly gathered the inputs and scattered the results for each call. Implicit gather/scatter is performed by providing the shapes and strides for all inputs/outputs from which the memory location of each element can be computed.

GPU CNN benchmarking

We benchmarked our gpuZNN implementation relative to the fastest available GPU frameworks [72] on an NVIDIA GeForce GTX Titan X (3072 cores, 1.0 GHz, 12 GB memory). Specifically, we benchmarked the MaxoutNet network with both 8 and 32 channels for three GPU frameworks: (a) gpuZNN, our modified version of ZNN [359]; Caffe [177], compiled with cuDNN v4; and Neon by Nervana [264], which outperformed Caffe and other alternatives on several recently published benchmarks [72]. Throughput was evaluated using 6 nm images presented with different sizes $(1024^2, 512^2, 256^2)$ in batches of size 32 (a constraint of Neon's GPU backend), while using max-pool layers with a stride of 1. This eliminates "sub-sampling" effects of max-pooling and simulates the same number of floating-point computations as a "dense" inference (not supported by Caffe or Neon).

The results of this benchmarking are presented in Figure 5-5. It is evident that gpuZNN provides the best throughput on both 8 and 32-channels for all image sizes: 1.8x and 2.9x faster than Neon and Caffe for 8 channels respectively, and 1.8x faster than Caffe for 32 channels. The 32-channel MaxoutNet could not be benchmarked on Neon as the memory usage exceeded the 12 GB available on the Titan X. The main reason for gpuZNN's improvement is its low-memory overhead FFTs that allows it to aggregate large sub-computation units and minimize the amount of calls to primitives of the GPU (each such call involves host-device memory transfers on the PCI-E bus). We note that the actual speed difference between 32 and 8 channels for gpuZNN is 2.6x (and not 32/8=4), which is an effect of the overheads of PCI-E host-device memory transfers that become more significant as the network becomes smaller.

A comparison of CPUs and GPUs

Table 5-6 shows the maximum throughput that we could achieve for XNN, gpuZNN and Neon for 8 and 32 channel CNNs. For 72-cores, we found that the best combination of multi-threading and multi-processing is 8 instances (2 per chip), where each is using 18 threads. One can see that for a network with 32 channels, the GPU is faster than the CPU. However, this is not the case for 8 channels, where the CPU is twice as fast as the GPU. This is because an 8 channel CNN implies less compute for the GPU and makes the PCI-E bus memory transfers more expensive.

5.4 Watershed

The next stage in the connectomics pipeline involves producing an over-segmentation of neuron candidates from the CNN membrane probability output. Over-segmentation ensures that no supervoxel straddles more than one true segment and is later resolved by agglomeration. Specifically, we apply a custom 3D implementation of the popular linear-time watershed algorithm that yields an 11x speed-up on previous implementations [174]. In general, the key idea is to take into account that probability maps are allowed to take only 8-bit values, which allows us to implement the priority queue as a set of 2^8 FIFO queues. The FIFO queues are implemented using simple arrays with an amortized cost of $O(1)$ for both *push* and *pop* operations. As a result, queue accesses are cache and pre–fetcher friendly. In addition, we reduce the memory overhead by an order of magnitude compared to OpenCV, which is an important factor for the watershed algorithm that exhibits low ratio of compute-to-memory.

5.5 Agglomeration

The agglomeration stage of our pipeline is based on Neuroproof [279], a state of the art tool for graph-based image segmentation. Serial optimizations and shared memory parallelization techniques were employed to enhance the performance of Neuroproof on our shared memory multicore system, with the results summarized in Figure 5-7.

5.5.1 Regional adjacency graphs

Agglomeration is performed on a higher-level representation of the labeled volume called a *regional adjacency graph (RAG)*. A RAG is a graph with a vertex for each distinctly labeled region, and an edge connecting each pair of adjacent regions. Each vertex within the RAG maintains several regional features such as histograms of membrane-probabilities and multiple image moments. These features are typically associative with respect to the agglomeration's merge operation, and thus enable the agglomeration procedure to operate directly on the RAG instead of the much larger input volume.

The RAG construction is the most expensive stage. Prior to our performance engineering efforts, it required 124 seconds on a 1024x1024x100 block and was responsible for 70% of the total time.

	Baseline (s)	**Fast 1-core** (s)	**Fast 4-core** (s)
Input/Output	11.2	10.5	6.7
RAG Construction	123.2	24.2	6.8
RAG Agglomeration	42.5	40.2	13.4
Total	176.9	75.0	27.0

Figure 5-7: Performance improvements to agglomeration stage.

Serial optimizations

Feature computation was optimized by providing a batched version the feature computation function, so that it could process multiple pixels at once. In addition, a significant optimization was for the image moment features. The ith image moment feature computes the sum of p^i for all pixels p in a region. The standard library `pow` function was used to compute these features for 4 moments, and was to blame for over 50% of the runtime in RAG construction. We modified the function computing moment features to perform iterative multiplication, and reduce the per-pixel-cost to 3 floating point multiplications. These modifications to feature computation resulted in a 2.5x improvement in serial runtime for RAG construction, resulting in a total runtime of 50 seconds.

Another class of changes involved eliminating unnecessary layers of indirection in matrix and hashtable access. The labeled input volume was loaded into an OpenCV `Mat` object and was accessed through a provided interface. We modified all accesses to operate directly on matrices underlying arrays to eliminate unnecessary indirection to the OpenCV library. This resulted in a further 1.6x improvement in RAG construction time, reducing the total runtime of RAG construction to 30 seconds.

The last major optimization was the design of a divide-and-conquer RAG construction algorithm that takes advantage of the associativity of region features. Given a 3D volume, the largest dimension of the volume was divided in half and the RAG was computed for each half recursively. These RAGs were then merged with their regional featured combined according to each of their associative update rules. The base-case of the recursion was coarsened so that the total volume would fit into an L2 cache of approximately 256 KB.

Parallelization

The divide-and-conquer implementation of RAG construction was especially amenable to efficient parallelization. In particular, the method of merging RAGs can be performed efficiently by thinking of the algorithm as a parallel merge sort with an augmented merge operation [77, Ch 27.3]. In other words, the RAG construction algorithm can be parallelized in the same fashion as merge sort by considering a RAG to be a sorted edge-list. We represent regions with a self-edge and store the region's computed features as metadata. After performing a merge of two RAGs' edge lists, the edge list is scanned in parallel to identify and merge duplicated edges. These steps can all be performed in parallel using logarithmic-depth algorithms. On 4 cores, the parallel RAG construction algorithm achieves speedup of 3.5x, reducing the time of RAG construction to 6.8s.

After RAG construction, RAG agglomeration analyzes edges to determine which node pairs to merge. We observed that most of the work was performed by a random-forest classification library within OpenCV that computes edge weights. To parallelize RAG agglomeration, we defer the computation of edge weights, and mark affected nodes and edges as *dirty* [279]. A dequeued edge is ignored if it is dirty or incident to a dirty node. When the priority queue is empty, the edge weights are all computed in parallel as a batch, and reinserted into the queue. This strategy resulted in 3x speedup for agglomeration, reducing its runtime from 40.2 to 13.4 seconds.

5.6 Merging

The state-of-the-art Spark-based system of Plaza and Berg [284] generates overlaps between adjacent blocks and identifies "merge pairs" from these overlaps. These merge pairs are identified from a set of complicated, hand-crafted heuristics, and the authors report that this is made difficult by the substantial number of edge cases. Instead, our merging algorithm leverages NeuroProof's fast random forest algorithm to identify merge pairs directly [254]. We further optimize this process by ordering and parallelizing computations in a way that maximizes utilization of the cache sub-systems, *e.g.* by processing small subsets that fit in the L3 cache of each chip.

Merge Pair Decisions. Merge pairs are determined for all adjacent blocks in the volume in two phases. First, the segments in the block boundary are agglomerated using our parallel NeuroProof. This phase is conservative, so only clear matches are merged. In the second phase, the algorithm generates a synthetic ground-truth by executing watershed and agglomeration on the block boundary. Then, it executes an optimization procedure based on the synthetic ground-truth: it merges a pair and computes the associated VI score relative to the ground-truth. If accuracy decreases, it aborts the merge and tries a different pair. Note that since most of the pairs are merged in the first phase, the second phase has fewer pairs to process and is substantially faster.

Combining and Relabelling. The merging procedure gives each voxel a new (final) label in the whole volume. To combine labels, we implemented a disjoint-set data structure, which clusters segments from different blocks according to the merge pair decisions. In a cluster-based system, this process is complicated by network transfers and failures, where in our case all intermediate data is immediately available in RAM or on disk, As an example, it takes just 4 seconds to combine labels of a 473 GB dataset – a speed which would be unobtainable on a network of hundreds of machines. Finally, the algorithm executes a parallel pass over all segmented blocks and relabels them according to clustering from the disjoint-set data structure. As this step is I/O bound, we utilize all of the drives to maximize I/O throughput.

5.7 Skeletonization

A skeleton is a one dimensional space-efficient representation of a 3D volume that runs along the segments's medial axis. There are a vast number of algorithms to compute skeletons for a variety of purposes in computer graphics [299]. We found that for connectomics a thinning algorithm would be most fitting because it is the fastest algorithm that still preserves the connectivity of the objects. Their algorithms remove all so-called *simple points*, voxels whose removal does not change the topology. Checking if a point is simple or not is non-trivial, especially when multiple voxels are deleted in parallel.

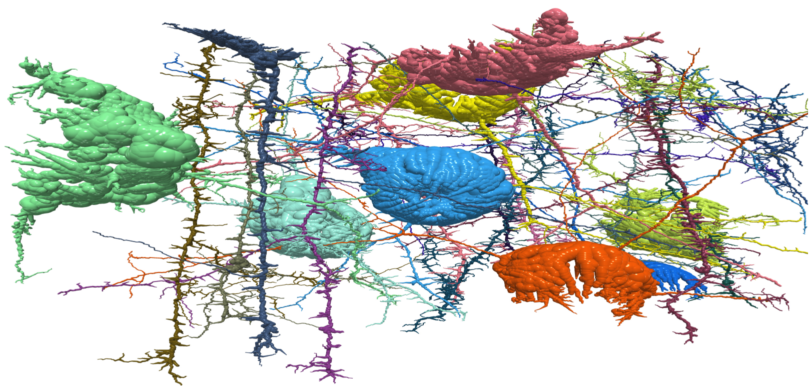


Figure 5-8: Skeletonization of 20 objects from the Kasthuri *et al.* dataset (as one large block) [191] (473 GB ≈ 100,000 cubic microns of cortex).

Our parallel thinning algorithm is implemented as a chromatically scheduled dynamic data-graph computation [183]. A grid graph represents the labeled 3D volume and a statically-computed distance-2 coloring is used to identify sets of voxels that may be processed in parallel.

Initially updates are scheduled on all surface points, which check if a given point satisfies the *simple point* criterion based on their 26-neighborhood. An update that deletes a point schedules an update dynamically for all non-deleted neighbors. Since there are only 2^{26} possible 26-neighborhoods, the simple point criterion can be precomputed. Our simple point criterion is based upon the 38 templates of [314] and ensures local connectedness. Since our algorithm uses chromatic scheduling, it is equivalent to a standard 8-subfield thinning algorithm and thus provably preserves the topology of the objects [25].

After skeletonization, we transform the 3D segments into tree graphs based on 26-connectivity. If cycles exist in the graph they are broken up arbitrarily, however, large cycles are detected and are reported for further analysis and error detection. After performing a one-pass pruning step the trees are outputted as .swc files and visualized using the neuTube software[104]. Although not scalable, the same algorithm can run on the entire volume instead of the per-block approach (Figure 5-8).

5.8 Pipeline performance

Machine learning consumes a large fraction of the computation in existing systems [284, 294], and was the main target of our scalability effort, so we spent time understanding its performance on both CPUs and GPUs earlier in the chapter. Here we focus on the combined performance of our pipeline elements. The main workhorse for our system testing is the previously described 4-socket shared memory machine equipped with 4 18-core Intel Xeon CPUs with 512 GB of RAM running Ubuntu 14.04.

We tested our pipeline on a 473 GB (3x3x30 nm resolution) electron microscopy dataset of mouse somatosensory cortex [190], containing 1850 pre-aligned 16,384 x16,384 (256 MB) 2D EM brain scans.

Figure 5-9 shows our total execution times for all of the pipeline stages. The total time to process the entire 473 GB EM dataset was **1.7 hours**, implying a throughput of **3.6 hours/TB**. Importantly, we have managed to improve the machine learning-based segmentation component to be on par with others, which has previously been a bottleneck

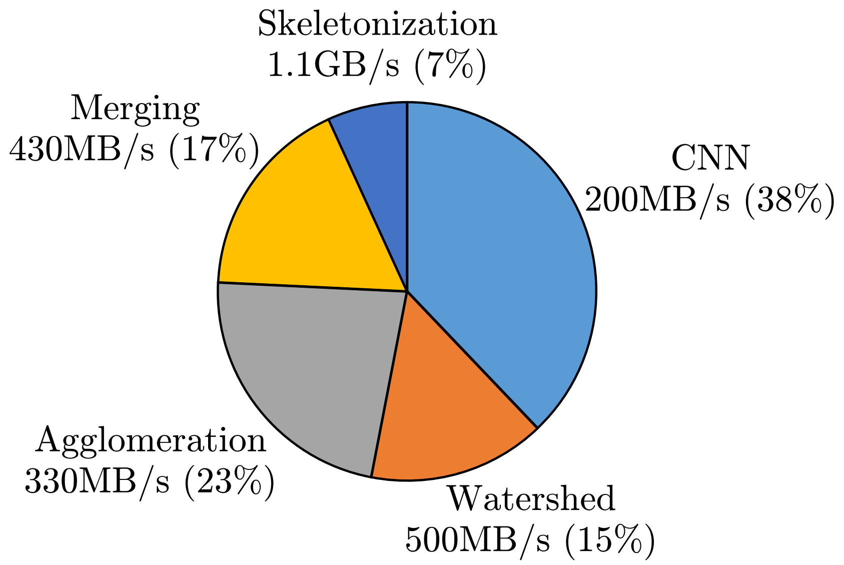


Figure 5-9: Proportion of the execution time spent on each stage.

in the throughput of connectomics pipelines [284].

To analyze the efficiency of using a 72-core system, we measured the time that it takes to execute our pipeline on a single core. The result is 1.11 s/MB total on a single core, compared to 0.013 s/MB on 72-core. This means that our pipeline achieves an 85x factor speedup, which is more than 72 because of the hyper-threading that allows us to generate 144 hardware threads on a single machine.

To achieve maximum throughput in the CNN phase, we execute 8 instances of XNN (2 per chip), where each instance uses 18 threads. This eliminates any NUMA side-effects that could introduce contention and bottlenecks. In the watershed phase, we simply spawn 144 instances since the implementation we have is a fast sequential code with low memory consumption (each instance <1 GB, and total memory is 512 GB). In the agglomeration, we execute 36 instances of the parallelized NeuroProof's code (9 per chip), where each instance uses 4 threads. A similar scheme is applied to merging and skeletonization.

5.8.1 Reconstruction accuracy

To evaluate the reconstruction accuracy of our pipeline we followed the benchmark described by Kaynig *et al.* [192]. We partitioned a total of 150 images (a region of the data called AC3 in [192]) at 2048x2048 and 3 nm resolution into three sets; 10 for training, 65 for validation and 75 for testing. Ground truth neuron segmentation was provided by expert neurobiologist annotators.

Our measure of segmentation accuracy is *variation of information* (VI) [253], that captures the statistical difference between two segmentations. In a typical segmentation there are split errors (neurons split erroneously) and merge errors (neurons merged erroneously), and the VI is an indicator to the extent of such errors in the data set. We benchmarked our 8-channel MaxoutNet CNN architecture (described in Section 5.3.1) and the state-of-the-art reconstruction of RhoAna described by Kaynig *et al.* [192] using the same evaluation benchmark on our test dataset described above. Our reconstruction obtained an accuracy score of VI = 1.6483, a substantial improvement over the VI = 1.99 score reported for

RhoAna (lower VI means improved reconstruction).

5.9 Lessons learned

In this section we discuss some of the general lessons learnt from our effort to reduce a high-throughput, big-data problem to the domain of a single shared-memory multicore system. Some of these may be known to the reader, but others were counter-intuitive, at least to us, and therefore worth explicit mention.

Software scalability reduces memory requirements. Writing multicore code is generally hard – it is far simpler to write a single-threaded program that avoids the associated concurrency "nightmares," and scales horizontally by launching a separate instance per core. While this may work for simple tasks, we observed that this approach can be seriously flawed for complex software. In the case of both our NeuroProof and CNN implementations, the large memory footprint of each instance caused the OS to frequently swap between memory and disk, thus increasing I/O bandwidth requirements. Moreover, scalability degraded due to L3 cache pollution – multiple instances spawned on the same 18-core chip populated the shared L3 cache with disjoint memory accesses. To tackle this issue, we engineered our software so each instance utilizes all 18 cores. For our 4-socket 72-core machine, 4 instances of this software were spawned, (one per socket) to avoid handling complex NUMA overheads [58, 94, 95, 102, 255], although it is possible that further work could allow a single instance to effectively utilize the full machine.

Disk I/O on a single machine is not problematic. Once computation is sufficiently optimized, disk-to-memory I/O for a single machine can become a bottleneck. This is often part of the motivation to migrate computation to a cluster of machines, and is a problem we encountered in the Watershed and agglomeration pipeline phases (where the disk I/O 100 MB/s read and 200 MB/s write were reached). Instead, we resolved this bottleneck by horizontally scaling our disk drives – data was sharded across a set of 5 drives, yielding 500 MB/s read and 1000 MB/s write for the system. Adding more disks to improve I/O bandwidth in this manner is far cheaper and simpler than migrating to a distributed cluster.

Cilk simplifies cache-aware multicore parallelization. A potential criticism of a multicore implementation approach is that we are simply shifting from one set of programming complexities (networking, failure detection and recovery schemes on a distributed system) to the unique challenges posed by multicore concurrency. Our experience is that effective multicore programming can be very straightforward. We make extensive use of the GCC-Cilk work-stealing scheduler to parallelize loop iterations, which removes the need to manually handle jobs/threads, encapsulate their environment (variable stacks) or schedule across multiple cores. In this way, multicore scalability only requires us to ensure that loop iterations (forming Cilk jobs) are cache and pre-fetcher friendly.

A GPU is not 100X faster than a CPU. It is widely believed in the machine learning community that a single GPU can perform orders of magnitude faster than a single CPU. This is supported empirically by popular machine learning packages – if one compares Caffe (bound to Intel MKL and Nvidia cuDNN libraries) execution times on GPU versus CPU, the speed-up is approximately 50 to 100-fold. Following our multicore performance engineering efforts, we observe substantially different results. On a single 18-core 2.4 GHz Haswell chip, our XNN execution is only 2-3x slower than both gpuZNN and the previous fastest CNN framework (Neon [264]) executed on a Titan X GPU. Moreover, XNN execution on a commodity 4-core Skylake chip (a standard desktop/laptop processor) was only 4-6x slower than this top-end GPU. These observations are consistent with the findings reported in [216], but here we support the claim on a modern CPU.

Multicores enable new efficient algorithms that are expensive on clusters. Cluster-based algorithms exhibit high network latencies, which forces them to avoid communication between machines. As a result, programmers are constrained to coding patterns that are "trivially parallelizable" in the context of clusters, since it is the only way to avoid high network costs and get scalability. However, this is not the case for multicores: inter-core communication is orders of magnitude less expensive than network cluster communication, which allows programmers to design more sophisticated approaches that involve complex communication patterns. For example, in our case, a multicore pipeline can use a sophisticated scheduler to split the work between cores: it can identify problematic data regions and schedule cores to process them without the need to worry about communicating the problematic data and setting the cores (this is a much more complex and expensive process on a cluster).

Programming for multicores is not harder than for a cluster. A common belief of Hadoop/Spark programmers is that coding for a distributed cluster is simpler than for a multicore, and therefore, shifting software to multicores is not worth the effort. However, our experience shows that this not true. A key point that needs to be emphasized is that coding for clusters is simple as long as the parallelization of the problem is "trivial". In other words, as long as one can break the problem to small parts without the need to use communication between these parts, then it is easy to do map-reduce style programming. However, if this is simple for the cluster, then it will also be simple for the multicore. This is because one can use threads to execute the small parts, and since there is no communication between these parts, there is no need to detect and handle race-conditions, which are the complex and hard part of the multicore programming. As a result, if programming cluster is simple, then programming multicore must be simple as well.

5.10 Conclusion

We presented a "proof of concept" connectomics pipeline that can extract a full skeletonization from an EM image stack in less than 4 hours on a commodity multicore machine and with a VI accuracy on par or better than any existing system. This has the potential to move the connectomics problem, a big-data research problem in the natural sciences, from the realm of distributed data and warehouse storage, to that of on-demand processing in labs across the world. Given current trends in multicore CPU and GPU architectures, we venture to predict that a single socket machine, perhaps with a single attached GPU card, will be able to provide a solution for many connectomics pipelines around the world.

The domain-specific results we report on Kashturi data-set [191], constitute a case-study on the role of multicore performance engineering in large-scale image processing pipelines. The lessons learnt from our experiences designing this system are of great import to those in the multicore and cluster computing communities. Through careful performance engineering, we show that a single commodity multicore machine can not only compete, but significantly outperform, existing CPU- and GPU- based clusters solving the same problem. This, of course, is not intended to advocate that all “big data” problems are best solved on a single multicore, but rather to serve as a reminder of the importance and dramatic benefits that can be obtained through multicore performance engineering.

5.11 Acknowledgements

Support is gratefully acknowledged from the National Science Foundation (NSF) under grants IIS-1447786 and CCF-1563880, and the Intelligence Advanced Research Projects Activity (IARPA) under grant 138076-5093555. In addition, we would like to thank the MIT SuperTech group for helpful discussions. In particular, Quan Nguyen, William Hasenplaugh, Bradley Kuszmaul, and Charles Leiserson. Finally, we would like to thank the Harvard connectomics group for reviewing our work and providing useful feedback. Specifically, Daniel Berger, Adi (Suissa) Peleg, Thouis R. Jones, Hanspeter Pfister, David Cox, and Jeff Lichtman.

Chapter 6

High-Throughput Image Alignment for Connectomics using Frugal Snap Judgments

This chapter presents a high-throughput image alignment pipeline for connectomics that employs the multicore algorithms Quilter and Stacker to perform 2D and 3D alignment respectively. As part of the optimization of this pipeline, this chapter introduces a technique for data-driven performance optimization called *frugal snap judgments* that is used to obtain more advantageous performance–accuracy trade-offs in Quilter. This work was conducted in collaboration with Brian Wheatman and Sarah Wooders.

Abstract

Accurate and computationally efficient image alignment is a vital step in the field of connectomics, which seeks to understand the structure of the brain through electron microscopy.

We introduce the algorithms Quilter and Stacker that are designed to perform 2D and 3D alignment respectively on petabyte-scale data sets from connectomics. Quilter and Stacker are efficient, scalable, and simple to deploy on hardware ranging from a researcher's laptop to a large-scale computing cluster. On a single 18-core cloud machine each algorithm achieves throughputs of more than 1 TB/hr and when combined produce an end-to-end alignment pipeline that processes data at a rate of 0.82 TB/hr — an over 10x improvement over previous systems. This efficiency comes from both traditional optimizations, and from the use of "Frugal Snap Judgments" to judiciously exploit performance–accuracy trade-offs.

A high-throughput image alignment pipeline was implemented and evaluated using the Quilter and Stacker algorithms. The performance was evaluated on a range of platforms including a common 18-core machine (Intel E5), a large 112-core machine (Intel Xeon Platinum), and a supercomputing cluster with 1600 cores. The pipeline achieves a throughput of 0.6–0.8 TB/hr on the 18-core machine, 1.4–1.5 TB/hr on the large 112-core machine, and 21.4 TB/hr on the supercomputing cluster with 1600 cores.

6.1 Introduction

Accurate and computationally efficient image alignment is vital within the field of connectomics [227, 226, 313, 298, 331], which seeks to study comprehensive maps of connections in the brain through the analysis of extremely-high resolution imagery obtained from electron microscopes. Advances in electron microscopy have enabled the acquisition of image data sets that capture both the small and large-scale features present in neural tissue. The resultant data-sets are quite large with a relatively small $1mm^3$ volume producing petabytes of data when imaged at $(3 \times 3 \times 30)$nm resolution. The scale of the acquired data necessitate the development of image processing algorithms that are both scalable and efficient.

The process of imaging a large physical volume at such high resolution does not, immediately, produce a single representative three-dimensional image. Instead, the volume to be imaged is physically sliced into very thin sections (approximately 30nm thick) which are then each imaged separately. Imaging a single two-dimensional section in its entirely is, unfortunately, also not possible since the size of each section vastly exceeds the limited field of view of the microscope. In order to image each section, therefore, the head of the microscope must physically move to scan over the entirety of a section each time capturing a single image tile of the section. These images must be aligned using 2D and 3D alignment algorithms in order to correct for the physical movements of the microscope and the deformations introduced during the slicing process [227].

This chapter presents a case study on the engineering of a high-performance image alignment pipeline for connectomics that makes efficient use of commodity multicore hardware whilst retaining both strong and weak scalability. The focus of our discussions shall be on the development of a high-performance variant of an existing, widely used, image alignment algorithm as implemented in software tools such as FijiBento [293] which is widely used within the connectomics research community [178, 312, 209, 311].

Designing an efficient alignment pipeline for multicore

Existing systems for performing image alignment in connectomics distribute fine-grained tasks over large compute clusters that have specialized hardware (GPUs) for accelerating performance-intensive tasks. Although there exist many, often lab-specific, alignment systems, they all tend to closely follow the methodology used in the FijiBento library [293] and the HHMI/Janelia Aligner project [306]. Each of these systems distributes work in stages where each stage is composed of one or more tasks for each image tile (approximately 8 MB) in the dataset. The intermediate results needed to combine the results of these tasks to perform 2D and 3D alignment are communicated either directly over the network or, more commonly, through a distributed filesystem such as Lustre [45] or pNFS [152]. In Section 6.2 we review the algorithms used in connectomics to perform 2D and 3D alignment and discuss the performance challenges that previous systems have encountered.

In Section 6.3 and Section 6.4 we describe the two multicore algorithms that form the backbone of this work's alignment pipeline. In Section 6.3, we introduce Quilter which is an efficient multicore algorithm for 2D alignment that avoids costly serialization and communication of intermediate results. Key to the design of Quilter is its efficient task ordering which allows it to reap the advantages of fast shared-memory communication of intermediate results whilst retaining the ability to scale to data sets that are much larger than the available memory of a single multicore. In Section 6.4, we introduce the Stacker algorithm which performs 3D alignment of sections. Stacker enables the alignment of a stack of sections in parallel by computing the pair-wise alignment of all adjacent sections concurrently. Although this strategy is often employed when performing affine alignment of sections, this method has not been used with the non-uniform elastic transformations that are needed to align imagery in connectomics.

Exploiting performance–accuracy tradeoffs

A simple way to increase the throughput of an image processing pipeline is to downsample the data. Downsampling the data is a tempting strategy for improving the performance of Quilter. For the vast majority of tiles in a section, aligning neighboring tiles using downsampled images does not change the computed relative alignment.

Our discussions with practitioners performing image alignment on connectomics datasets, however, revealed a wariness towards downsampling due to its potential to introduce errors. Their rationale is twofold: (a) the alignment errors introduced in 2D alignment, while small, can accumulate when performing stitching of a large section resulting in unnatural deformations that are difficult to correct; and, (b) even small misalignments are sufficient to alter fine-details in certain neuronal structures such as the spine necks of dendrites. Indeed, small differences in alignment can impact later stages of the connectomics pipeline; e.g., small misalignments of 4–8 pixels can cause neuronal objects to be incorrectly split or merged [227].

The potential performance advantages of downsampling, however, were too alluring to pass up. Thus, we developed a technique called *frugal snap judgments* that allows us to obtain more advantageous performance–accuracy trade-offs. Figure 6-1a and Figure 6-1b show the throughput and accuracy obtained for 2D alignment when using frugal snap judgments (FSJ). Using frugal snap judgments, our system learns to identify when the result of the fast alignment algorithm is likely to match the result of the more-accurate code. This enables us to achieve a 5x improvement in throughput without introducing significant alignment error.

In Section 6.5 we introduce *frugal snap judgments* and explain how we applied it to the image alignment problem to obtain more advantageous performance–accuracy trade-offs. Using frugal snap judgments, our system learns to identify when the result of the fast alignment algorithm is likely to match the result of the more-accurate code. The learned criteria is not based upon an objective notion of correctness (which would be costly to compute). Instead, it is based upon intermediate results generated by the fast alignment code that are analyzed to extract a measure of reliability.

Performance evaluation of Quilter and Stacker

Section 6.6 presents an end-to-end performance evaluation of the image alignment pipeline we developed using Quilter and Stacker. We investigate the performance of Quilter and Stacker on a set of four computing platforms that range from a modestly sized desktop machine to a large compute cluster with thousands of cores. The efficient design of Quilter and Stacker combined with the judicious exploitation of performance–accuracy trade-offs using frugal snap judgments allows us to obtain state-of-the-art performance while using only commodity multicore hardware. Our pipeline achieves 0.6–0.8 TB/hr throughput on a commodity 18-core Intel Xeon CPU E5-2666; 1.4–1.5 TB/hr throughput on a large 112-core Intel Xeon Platinum 8180 (2.5GHz); 7.5 TB/hr on 5440 AMD Opteron 6274 processors (2.2GHz); and, 21.4 TB/hr on 1600 Intel E5-2666 cores.

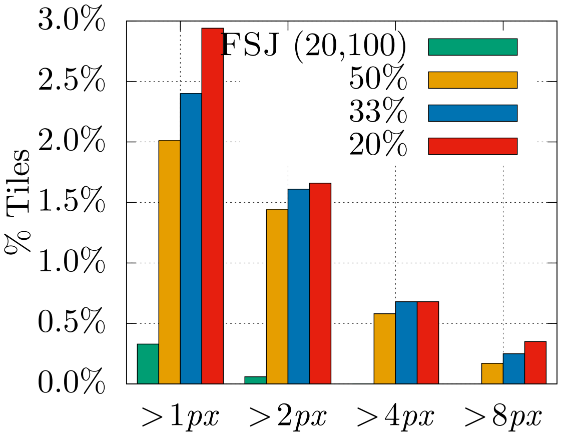
Contributions

A summary of the contributions are as follows.

- An efficient 2D alignment algorithm called Quilter with a low memory high-watermark that does not perform redundant file I/O or recomputation.

Algorithm	Runtime	Throughput
100%	180 minutes	0.18 TB/hr
50%	86 minutes	0.38 TB/hr
33%	49 minutes	0.67 TB/hr
25%	30 minutes	1.1 TB/hr
20%	27 minutes	1.2 TB/hr
FSJ(20%, 100%)	32 minutes	1.03 TB/hr

(a) Alignment throughput.



(b) Alignment errors.

Figure 6-1: Impact of downsampling on throughput and accuracy. Figure 6-1a shows runtime and throughput of 2D alignment on the *mouse50* dataset: a 550GB dataset composed of 65, 000 tiles. Figure 6-1b illustrates alignment errors on *mouse50* greater than 1px, 2px, 4px, and 8px relative to the algorithm run on full-resolution images. The algorithms 50%, 33%, and 20% downsample to 1/2, 1/3, and 1/5 respectively. FSJ(20%, 100%) downsamples to 1/5, but recomputes at full-resolution when detecting a likely error.

- A horizontally scalable 3D alignment algorithm called Stacker that supports affine and elastic transformations.
- We introduce the optimization technique of *frugal snap judgments* and apply it to improve the performance of Quilter by 3–5x with no significant accuracy loss.
- System evaluation of Quilter and Stacker on three datasets ranging in size from 550GB to 38TB, and on four computing platforms including a single 18-core workstation and a supercomputing cluster with thousands of cores. The evaluation illustrates end-to-end performance, vertical/horizontal scalability, and strong/weak scaling.

6.2 Alignment algorithms used in connectomics

In this section, we provide necessary background on the general structure of algorithms used to align large EM data sets in connectomics. The primary purpose of this discussion is to provide a basic understanding of the key operations that are performed during alignment when using FijiBento (and related) systems. Additionally, we will discuss some performance challenges related to memory-limitations and overheads due to data serialization and file/network I/O.

Image data format

Let us briefly describe the format of the images provided by the microscope to make our later back-of-the-envelope calculations concrete. The microscope provides a set of image tiles where each image is 2724×3128 pixels. Each pixel in an image represents a $(3 \times 3 \times 30)$nm physical volume. We assume that the set of tiles that compose a 2D section fall within a bounding rectangle whose length and width differ by a constant factor; and, we assume that the average tile overlaps with ≈ 8 neighbors. These assumptions are realistic guarantees that are ensured during the image acquisition process.

2D alignment algorithm

The first step of the alignment pipeline constructs a 2D *section* from the set of image tiles obtained by imaging a single physical slice of tissue. During the imaging process each image tile acquired is associated with an approximate location in the plane, and adjacent tiles are imaged such that there is a small amount of overlap. This overlap allows for the 2D alignment algorithm to identity common landmarks in adjacent tiles that are then used to precisely determine the tiles' position relative to eachother. After the relative alignments are computed, optimization is performed to find the tile locations that minimize the total energy of a system in which each adjacent tile pair is connected by a spring with rest position determined by the tile pair's relative alignment.

Algorithm 2 2D image Alignment

Result: locations of each tile in a single global space.

```
1   read in metadata file
2   create list $P$ of all pairs of possibly overlapping tiles
3   all_local_offsets = $\emptyset$
4   for $p \in P$
5       load_image(p.first)
6       load_image(p.second)
7       kp_1 = getKeyPoints(p.first)
8       kp_2 = getKeyPoints(p.second)
9       matched_kp = matchKeyPoints(kp_1, kp_2)
10      local_offset = find_best_offset(matched_kp)
11      all_local_offsets[p] = local_offset
12  global_offset = combine_offsets(all_local_offsets)
```

Algorithm 2 illustrates the structure of the 2D alignment step. The input to Algorithm Algorithm 2 is a metadata file that includes the location of each image tile on disk, and the approximate coordinates of each tile in the plane. These approximate coordinates induce a poor alignment, but are sufficient to determine the pairs of tiles that overlap. The image data for each tile is often compressed to reduce file and network I/O. A common format is JPEG2000[91] which obtains up to 10x compression ratios on connectomics data sets. Generating keypoints is commonly performed using the SIFT algorithm[235]. Other algorithms such as SURF[17] and ORB[296] are sometimes used in place of SIFT, but their use is less common. Keypoints in the two images are matched by finding the nearest neighbors in the SIFT feature space, and filtered using the ratio of difference heuristic described in [235]. Next, a random sampling and consensus algorithm (RANSAC) [110] is used to find a coordinate transformation (in the case of 2D a rigid translation) that is consistent with the maximum number of matched keypoints.

Design considerations for 2D alignment

A variety of designs were considered for the 2D alignment algorithm that vary in terms of their memory and communication requirements. Our goal in the design of Quilter, which we describe in Section 6.3, was to build a system that would: (a) read each image tile only once to save on extra I/O costs; (b) do not compute the same thing twice to save on extra compute costs; (c) do not serialize intermediate results to disk, to save on extra I/O costs; and, (d) have projected memory requirements that allow a single multicore to align a human brain.

To illuminate the design challenges addressed by Quilter, we shall discuss a few natural alternative designs and explain their shortcomings.

All-I/O and All-Mem

First, let us consider two extreme approaches: All-Mem which stores all images and intermediate results in-memory; and, All-I/O which keeps only a single pair of tiles in-memory and writes intermediate results to disk.

An All-Mem algorithm loads all images from disk and then computes the 2D alignment while keeping all images and intermediate results in-memory. All-Mem requires memory proportional to the size of the area being aligned which precludes its use for large sections. The advantages of All-Mem are that it only reads each image once, and does not need to perform recomputation. Thus, All-Mem satisfies (a)–(c), but fails to satisfy (d).

An All-I/O algorithm maintains a constant memory footprint by performing redundant computation and file I/O. For each pair of overlapping image tiles All-I/O reads both images from disk, computes the SIFT keypoints for both images, and uses those keypoints to find the tile pair's relative alignment. This relative alignment is written to disk, and then the images along with the computed keypoints are discarded. All-I/O must read and compute keypoints for each image approximately 4 times — increasing compute and I/O costs by 4x. Thus, All-I/O satisfies (c) and (d), but fails to satisfy (a) and (b).

Inter-Mem and Inter-I/O

Let us now consider two variations, Inter-Mem and Inter-I/O, that avoid recomputation by caching the keypoints computed for each image. The Inter-Mem algorithm reads each image and generates keypoints that are then cached in-memory. Then the relative alignment of all adjacent tiles are computed using the previously computed keypoints. The Inter-I/O algorithm operates analogously, but it caches keypoints on disk instead of in-memory. For a typical tile of size 8MB, the size of the cached keypoints is between 1.2–3.1MB[1]. As such, the Inter-Mem algorithm is generally more memory-efficient than All-Mem. Inter-I/O, however, is generally less efficient than All-I/O for two reasons: (1) images are often read from disk in a compressed format that can be 10x smaller than the uncompressed data; and (2) the Inter-I/O algorithm must read a tile's keypoints 4 times, on average, to align it to all of its neighbors. As such, Inter-I/O performs less computation than All-I/O by avoiding recomputation, but does so at the expense of extra I/O to access cached keypoints.

[1]A data table is provided in Figure 6-12 that contains the information needed for this back-of-the-envelope calculation

3D alignment algorithm

After the 2D alignment step, the now-constructed sections are aligned to form a 3D image that represents the imaged volume. A variety of processes during the imaging process can introduce distortions in the 2D sections that need to be corrected during 3D alignment. The physical cutting of the volume is the likely cause of most distortions, but other processes such as the transport of the very-thin sections via water-bed and non-uniform expansion/compression of the tissue due to environmental conditions can play a role as well. Regardless of the cause, these distortions must be corrected in order to obtain a representative 3D image for the sample.

The 3D alignment algorithm approximates the function mapping the coordinates of one section into another using a collection of affine transformations. Specifically, a triangle mesh overlays each section and a barycentric coordinate transformation [157] maps the points within each triangle into the coordinate space of adjacent sections.

Typical systems, such as those in FijiBento, for performing 3D alignment operate by dividing the volume into smaller 3D blocks and perform iterative optimization of the triangle mesh. Corresponding points between adjacent sections are identified (using a procedure similar to that used in 2D). Then, an iterative optimization procedure adjusts the triangle meshes in the volume to minimize the distance between corresponding points.

A disadvantage of this approach is that the entire volume must be aligned more-or-less simultaneously. The addition or removal of a single new section can necessitate the recomputation of the alignment for the entire volume. This can be especially problematic when one fails to identify a single bad section prior to investing the computing resources to optimize a large volume. Furthermore, this approach precludes the interleaving of 2D and 3D alignment in a principled manner — since the computed 3D alignment for a single section depends on all other sections.

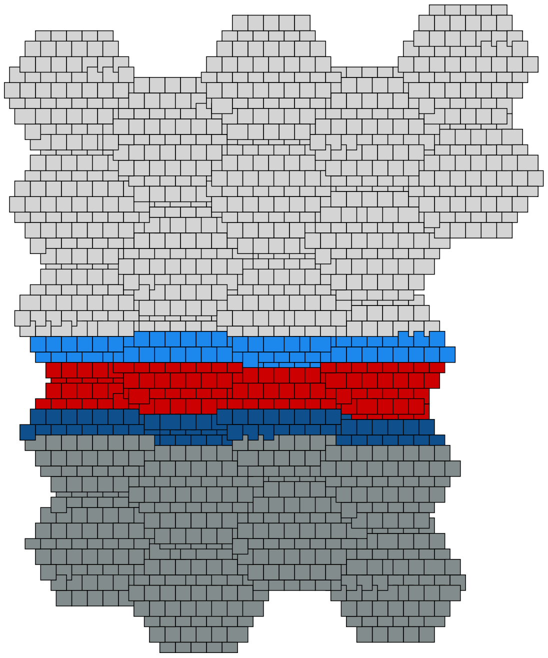
6.3 Quilter algorithm

This section describes Quilter: a 2D alignment algorithm for stitching very large mosaics in-memory. Quilter employs careful task ordering to bound its memory use while avoiding unnecessary file-I/O and recomputation. This enables Quilter to reap the performance advantages of in-memory computing even for very large mosaics. Quilter can process a 2D cross-section of an entire mouse brain with under 50 GB of memory, and process a 2D cross-section of human brain with less than a 1 TB memory.

Line-sweep ordering of tasks

The essential idea of Quilter is to order tasks based upon a 1D line-sweep through the section being 2D aligned. At any given point in the execution, the image data and intermediate results for tiles that are touched by the line-sweep and those tiles' neighbors are kept in-memory. The data retained for a tile is released once the line sweep has advanced beyond all of a tile's potential neighbors.

Figure 6-2 illustrates an in-progress execution of Quilter using this line-sweep task-ordering. Quilter begins by computing, for each tile, a set of overlapping neighbors and sorts all tiles by the y-coordinate of their bounding box's bottom-left corner. The initial set of tiles processed by Quilter consists of tiles whose bounding box overlaps with a horizontal slab extending along the bottom of the section. For each tile being processed, all of its



	Done	In-Memory
◻	✗	✗
🟦	✗	✓
🟥	✗ → ✓	✓
🟦	✓	✓
◼	✓	✗

Figure 6-2: An illustration of a Quilter execution in-progress. Tiles are shaded red if being processed, blue if it's keypoints are in-memory, and light/dark gray if it's untouched/done.

neighbors that are not already in-memory are read from disk. Quilter then computes pairwise alignments between selected tiles and all of their overlapping neighbors. Quilter then progresses by increasing the position of its horizontal slab to select a new set of tiles to process. This new slab will contain the old neighbors of the previous slab, which are already in memory. Before local alignment is computed, the slab's new neighbors are loaded into memory. Once the local alignment is finished the first slab's data can be released, the slab moved up and the process repeated.

After computing the pairwise alignments of all tiles in the section, Quilter solves the optimization problem described in Section 6.2 to position tiles in the plane using a loss function determined by the local relative alignments between tile pairs. This optimization problem is formulated and solved in parallel by treating it as a data-graph computation and employing the techniques from [184].

Memory requirements of Quilter

We shall now analyze the memory and I/O required by Quilter to align a section using the data from Figure 6-12.

Quilter performs I/O to read compressed image data and to write its final result to disk. For a 10cm^2 section, Quilter reads 100TB of compressed image data from disk. Quilter writes a list of tile ids and 2D offsets to disk which are negligible in size (less than 64 bytes per tile). The total memory required by Quilter depends upon the maximum size of its working set. For each tile in the working set we store the tile's uncompressed image data and all of its keypoints. The maximum size of the working set is approximately 3 rows of tiles. As such, the total memory required to process a 10cm^2 section using Quilter is approximately 800GB. A comparison of Quilter to the methods described in Section 6.2 can

Method	Memory	Total I/O	I/O Ops
All-Mem	1000 TB	100 TB	130 million
All-I/O	0.5 TB	400 TB	520 million
Inter-Mem	160 - 400 TB	100 TB	130 million
Inter-I/O	0.5 TB	640 - 1600 TB	780 million
Quilter	0.8 TB	100 TB	130 million

Figure 6-3: Memory and I/O Characteristics of 2D alignment algorithms on a $10cm^2$ section imaged at $(3 \times 3 \times 30)$nm resolution with tile size 2724×3128 pixels.

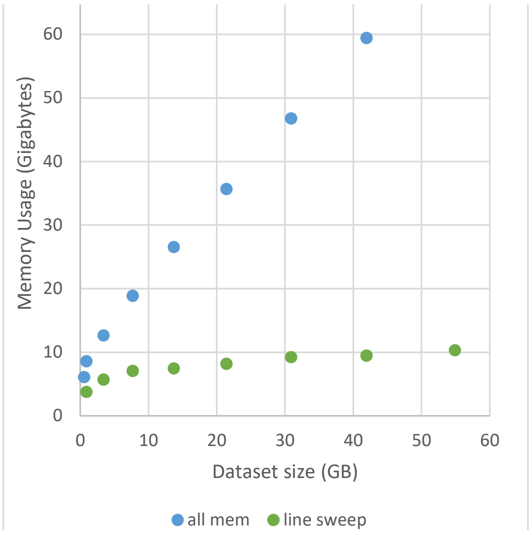


Figure 6-4: Memory high-water mark of Quilter compared to all-mem as the data set grows. Data obtained by extracting regions from regions from *human100*.

be found in Figure 6-3.

The memory requirements of Quilter relative to All-Mem were measured empirically and plotted as a function of dataset size in Figure 6-4. For this experiment, we ran All-Mem and Quilter on progressively larger subsets of a large 2D section of $40,000$ tiles. These subsets were circular regions centered in the middle of the 2D section with varied radius. As expected, the memory requirements of Quilter scale proportionately to the diameter of the region being aligned, and All-Mem scales proportionately to the area.

6.4 The Stacker algorithm

This section describes the Stacker 3D alignment algorithm. Stacker operates on pairs of adjacent sections that have been 2D-aligned using Quilter. Stacker supports both affine transformations and non-affine "elastic" transformations which enable Stacker to compute high-quality 3D alignments that correct for distortions introduced by errors during sample preparation or imaging.

Our discussion will focus on two aspects of Stacker that are especially relevant to the alignment pipeline described in this chapter: (a) its ability to align adjacent sections independently in-parallel, and (b) the memory requirements of Stacker when operating on large sections.

Independent alignment of sections

Stacker is designed to align pairs of adjacent sections independently using composable transformations. Given a pair of sections (S_{i-1}, S_i), Stacker computes a coordinate transformation mapping points in S_i to points in S_{i-1}. This coordinate transform is represented using a hexagonal triangle mesh that overlays section S_i. Each triangle vertex in S_i has a corresponding *transformed vertex* in S_{i-1} whose position is computed by Stacker during 3D alignment. After 3D aligning S_i to S_{i-1} a point in S_i can be mapped to a coordinate in S_{i-1} by finding the triangle containing that point, and performing a barycentric coordinate transformation[157] to obtain that point's location in the corresponding triangle within S_{i-1}.

Given a stack of sections $S_1, S_2, \ldots, S_k$ Stacker computes the pairwise alignment of $(S_1, S_2), (S_2, S_3), \ldots, (S_{k-1}, S_k)$ independently. To map all sections into the global coordinate space of a single 3D volume, the relative alignments are combined using function composition which can be accomplished efficiently by applying the barycentric coordinate transformations upon the vertices of the triangle mesh itself.

Memory requirements of Stacker

Stacker is designed to align a pair of sections on a single multicore. As such, a natural concern is that Stacker will be unable to align very-large sections efficiently due to the limited available memory of a single machine. Fortunately, the memory requirements of Stacker are actually quite modest and allow it to scale all the way to the human brain while only requiring about 4TB of memory.

A good estimate of the memory requirements of Stacker can be obtained via a back-of-the-envelope calculation. For each image tile of size $\approx$ 8MB, Stacker requires memory for approximately 100 corresponding points and 12 triangles. Each corresponding point, along with its SIFT feature vector, is represented using 156 bytes, and each triangle requires 128 bytes. Each section of human brain ($10cm^2$) is composed of approximately $125,000,000$ tiles, and so the total memory requirements of Stacker per-section is approximately 2TB. Since Stacker stores data for 2 sections while computing a pairwise alignment, Stacker requires about 4TB of memory to perform 3D alignment on sections of human brain.

Presently, the volumes being aligned in connectomics are substantially smaller than those of the human brain. For the "grand challenge" of reconstructing an entire mouse brain which is 100x smaller by volume than the human brain, the same back of the envelope calculation shows that Stacker requires only about 50GB of memory.

6.5 Frugal snap judgments

This section describes a technique called *frugal snap judgments* (FSJ) that is used to accelerate the performance of Quilter by a factor of 3-5x without any appreciable loss of alignment accuracy.

Design of FSJ in Quilter

Let us now describe how frugal snap judgments are applied to improve the performance of Quilter. Frugal snap judgments are used to optimize the algorithm used to compute

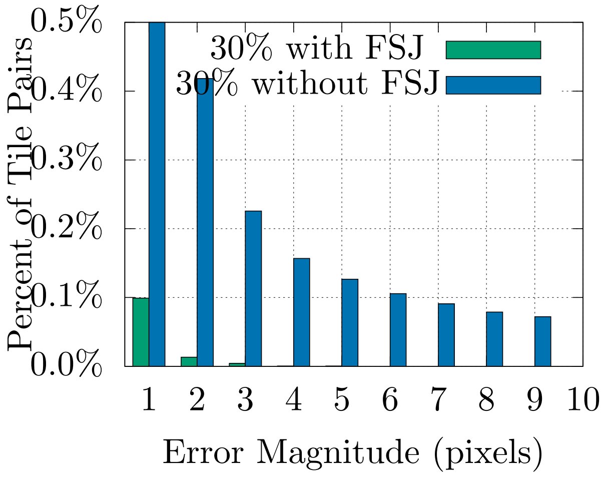


Figure 6-5: Error with and without FSJ on *human100* relative to the alignment Quilter produces with no downsampling.

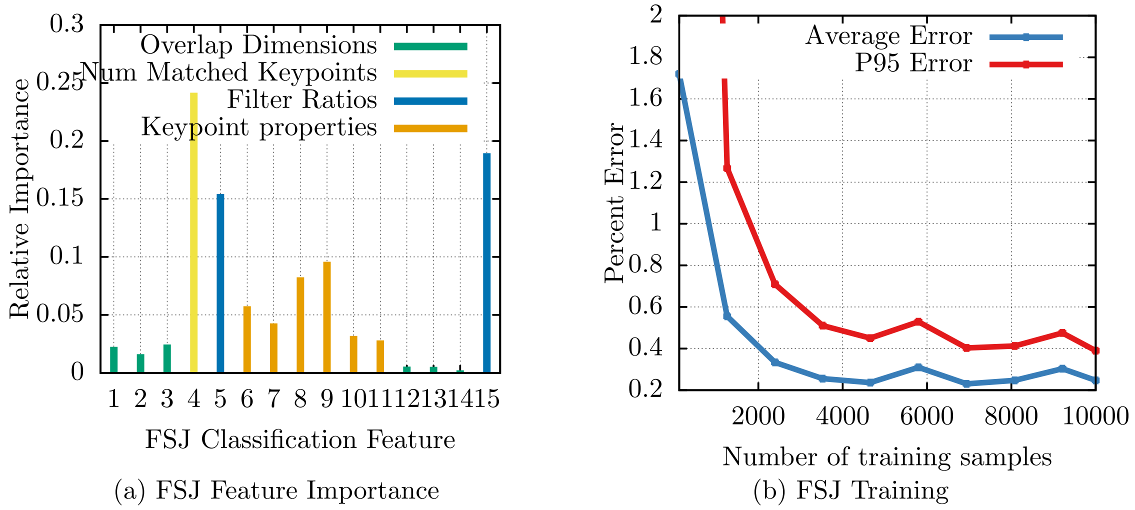


Figure 6-6: Figure 6-6a shows the relative importance of variables used by FSJ on *human100* dataset. Variables are numbered as follows: 1–3 and 12–14 are overlap area/length features before and after fast-path alignment; 5,15 ratios of keypoints filtered during matching; 6–11 aggregate statistics of overlap region; and, 4 is number of matched keypoints. Figure 6-6b illustrates FSJ classification error during training.

corresponding keypoints between two overlapping tiles.

The fast-path algorithm for computing pairwise tile alignments operates on downsampled tile images at 30% resolution. For approximately 2% of tiles and 0.4% of all tile pairs, these modifications result in a less accurate algorithm relative to the original code path operating on full-resolution images. Yet, for the remaining tile pairs the fast-path computes a result that matches the slow-path within a tolerance of 1-2 pixels.

In order to detect when the result of the fast-path is unreliable, we employ random-forest classification over feature vectors that summarize the intermediate results of the fast-path algorithm.

We employ feature vectors of dimension 15 which include the following information. The area, width, and height of the overlapping region between a pair of tiles, the fraction of keypoints matched, the fraction of matched keypoints that are filtered by RANSAC, the total number of filtered keypoints, and aggregate statistics from the filtered keypoints. Additional details are provided in Figure 6-6a. These feature vectors are small and inexpensive to compute since they depend on the intermediate results of the fast-path rather than directly depending on raw pixel data in the image tiles.

Training our fast-path detector

Data to train the detector is obtained through random sampling of overlapping tile pairs from the stack and executing the fast and slow path codes on the same input. The relative offsets between the tiles computed by the fast and slow path are compared and considered matching if they differ by less than 1px. We extract a feature vector from the fast-path result and insert it into either the training or testing sets.

During training, a *false positive* occurs when the detector incorrectly predicts that the fast/slow codes will match, and a *false negative* occurs when it incorrectly predicts that they will disagree. A well-trained detector will minimize the false negative rate while strictly constraining the false positive rate.

Figure 6-6b illustrates the evolution of the false-positive rate while training the FSJ classifier. A 95th percentile confidence bound on the false-positive rate is estimated, using the testing set, during training and used as a stopping criterion. For our data sets, we typically stopped training once the 95th percentile confidence bound on the false-positive rate fell below 0.4%. The false-negative rate of our FSJ classifiers varied between $\approx 4-10\%$ depending on the resolution of the fast path and the dataset. Training time does not depend on the size of the dataset, and took between 10-20 minutes on an 18-core Intel Xeon CPU (E5-2666 v3, 2.9GHz).

After training on the *human100* data set using 30% resolution images in the fast path, we achieved an out-of-bag error of $6.7e^{-2}$, a 95th percentile confidence false-positive rate of 0.4%, and a false negative rate of 6%. The relative variable importance scores for the random forest classifier are provided in Figure 6-6a.

Figure 6-5 shows the 2D alignment errors of Quilter when using FSJ on a set of 4 sections that were not used during training. When using FSJ, there are nearly no errors greater than 1 pixel, and 0 errors greater than 5 pixels.

6.6 System evaluation

This section provides end-to-end performance results for the alignment pipeline built using the Quilter and Stacker algorithms on three datasets across four computing platforms.

Dataset	FSJ	Platform	Hardware	Wall-clock runtime	Throughput
mouse200	FSJ30	*AWS Cluster*	200 8-core AWS C4 Instances (1600 cores)	5.6 minutes	21.4 TB/hr
mouse200	FSJ30	*LLSC Cluster*	170 Opteron nodes (5440 cores)	16 minutes	7.5 TB/hr
mouse200	FSJ30	*Large Multicore*	112-Core Intel Xeon Platinum 8180	80 minutes	1.5 TB/hr
human100	FSJ30	*Large Multicore*	112-Core Intel Xeon Platinum 8180	26.7 hours	1.4 TB/hr
mouse50	FSJ30	*Common Multicore*	18-Core AWS C4 Instance	49 minutes	0.67 TB/hr
mouse50	FSJ20	*Common Multicore*	18-Core AWS C4 Instance	40 minutes	0.82 TB/hr

Figure 6-7: End-to-end performance results for whole alignment pipeline executing both Quilter and Stacker. In the FSJ column, FSJ30 and FSJ20 indicate that the fast path employed by FSJ used 30% and 20% resolution images respectively.

2D Alignment Method	Runtime	Throughput
FijiBento	362 minutes	0.091 TB/hr
Quilter Full Resolution	180 minutes	0.18 TB/hr
Quilter FSJ(20,100)	32 minutes	1.03 TB/hr

Figure 6-8: Performance comparison of FijiBento and Quilter for 2D Alignment on the *Common Multicore* platform.

Experimental setup

A summary of the software, datasets and hardware used in our evaluation are as follows.

Software. The alignment pipeline composed of Quilter and Stacker was implemented as a C++ software library parallelized using Cilk Plus [36, 219] and the Tapir [304] branch of the LLVM [206, 207] compiler (version 6)[2]. The following software libraries were used: OpenCV v3.2.0 [47], OpenJPEG v3.2.0 [91], and Google protocol buffers [136].

Datasets. Three different data sets were employed in our evaluations: *Mouse50*, *Mouse200*, and *Human100*. *Mouse50* is 550GB dataset composed of 50 sections and 65,000 image tiles. *Mouse200* is a 2TB dataset composed of 200 sections and 200,000 image tiles. *Human100* which is a 100-section 38TB dataset.

Hardware. Our evaluations employ four different computing platforms to evaluate runtime performance: *Common Multicore*, *Large Multicore*, *LLSC Cluster*, and *AWS Cluster*. Figure 6-7 details the hardware for each platform, and additional details are provided in Section 6.7.

Multicore performance of Quilter and Stacker

Let us first analyze the efficiency of Quilter on the *Common Multicore* system relative to FijiBento. For this experiment, we used the *mouse50* dataset. Both Quilter and FijiBento compute equivalent 2D alignments, but FijiBento has additional overheads due to the serialization of intermediate results to disk. In Figure 6-8, we see that Quilter out-performs *FijiBento* by a factor of 2x when operating on full-resolution data. The use of frugal snap judgments in Quilter (Quilter FSJ(20,100) in Figure 6-8) provides a further performance boost of 5.6x, and causes Quilter to outperform FijiBento by a factor of 11.3x.

The performance of Stacker on the *Common Multicore* platform is similarly efficient. To 3D align the *mouse50* dataset Stacker required only 8 minutes of compute time on

[2]Available from `http://cilk.mit.edu`

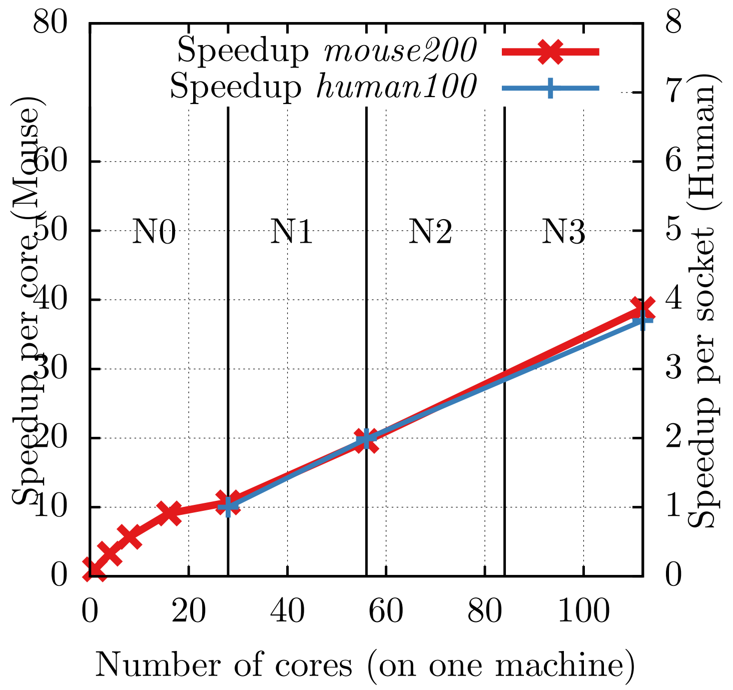


Figure 6-9: Vertical scalability. Reports speedup obtained when executing 4 sections of *mouse200* and 4 sections of *human100* on the *LargeMulticore* platform. The *mouse200* scalability is relative to a 1-core executing using the left-axis of the plot. The *human100* scalability is relative to a 1-socket execution using the right-axis. The mapping of cores to sockets is provided via the $N0, N1, N2, N3$ labels.

Common Multicore. The total runtime to 2D and 3D align *mouse50* was 40 minutes as shown in Figure 6-7.

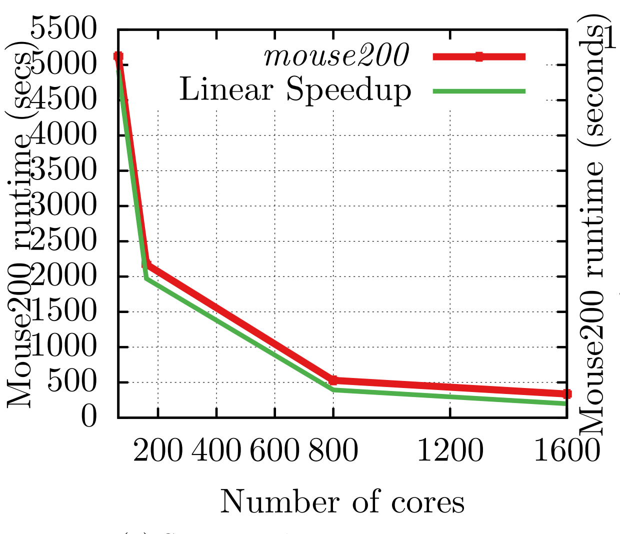
Next, let us analyze the performance of Quilter and Stacker on the *Large Multicore* platform to evaluate its scalability on a single machine with a larger number of processing cores.

Figure 6-9 illustrates the speedup achieved on the *Large Multicore* platform when aligning 4 sections of the *mouse200* and *human100* dataset. The left y-axis shows the speedup achieved on the *mouse200* dataset relative to a serial execution. On *mouse200* approximately 10x speedup is achieved on a 28-core (1 full socket) execution relative to a serial execution. The sublinear speedup is attributable, primarily, to aggressive downclocking of the cores on the socket when using AVX instructions [346]. On 112-cores (4 sockets), the speedup achieved on *mouse200* relative to a serial execution is approximately 39x. For the *human100* dataset we report the scalability relative to a 1-socket execution using the right y-axis.[3] Similar to *mouse200*, Each additional socket provides near-linear improvements in performance when aligning the *human100* dataset.

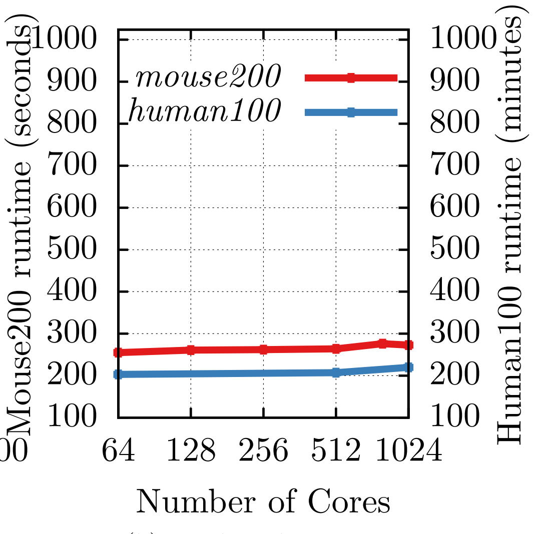
Scaling across multiple machines in a cluster.

Now we evaluate the scalability of Quilter and Stacker when executing the pipeline across many multicores in a computing cluster. For these experiments we use the *AWS Cluster*

[3] A serial execution of a section of *human100* is unduly time-consuming.



(a) Strong scaling.



(b) Weak scaling.

Figure 6-10: Strong and weak scalability on the *AWS Cluster* platform on the *mouse200* and *human100* datasets.

platform to align the *mouse200* and *human100* datasets.

The strong scaling of the pipeline is shown in Figure 6-10a. For this experiment, when running the pipeline on N nodes we divide the *mouse200* stack into $200/N$ blocks. A block is first processed by a single task that runs Quilter and Stacker back-to-back as a single shared-memory process on the sections within a block. Next, $N-1$ tasks are created that run Stacker on pairs of sections that border adjacent blocks. On the *mouse200* dataset and the *AWS Cluster* platform we observe near-linear strong scaling.

The weak scaling of the alignment pipeline is shown in Figure 6-10b. For the weak scaling experiment we ensured that the work-per-node was constant by scaling the number of sections being aligned proportionately to the number of machines used to execute the pipeline. We observe no appreciable decrease in efficiency-per-node when scaling from 1 to 128 machines where each machine is an 8-core Intel E5 node.

End-to-end system experiments

Figure 6-7 illustrates the absolute runtime obtained on the *mouse200*, *human100*, and *mouse50* data sets on the *Common Multicore*, *Large Multicore*, *LLSC Cluster*, and *AWS Cluster* systems. Due to the difficulty of moving the full *human100* dataset to different platforms, we only ran a full end-to-end test for this dataset on the *Large Multicore* platform. The highest throughput was achieved on the *AWS Cluster* platform which was able to align data at a rate of 21.4 terabytes per hour using 1600 cores.

6.7 Computing platforms and datasets

This section provides additional details on the computing platforms and datasets employed to evaluate Quilter and Stacker in Section 6.6.

Common Multicore is an 18-core 2-way hyperthreaded Intel Xeon CPU (E5-2666 v3, 2.9GHz) available as a 4th-generation compute-optimized machine from amazon web ser-

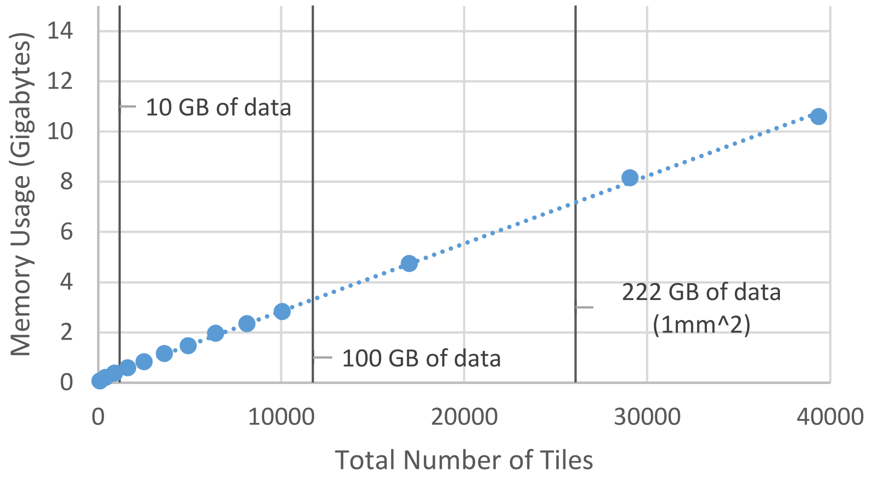


Figure 6-11: Memory high-water mark of Stacker vs. number of tiles aligned.

vices which has 64GB of memory and runs Ubuntu 14.04 on Linux Kernel 3.13.0-106. Amazon EFS [6] (elastic filesystem) is used to store image data files. Amazon EFS provides performance tiered to the size of the mounted volume. Based on these tiers and our mounted volume size of 2.8TiB our maximum burst IO throughput was 100 MiB/s during our experiments on this platform.

Large Multicore is a 112-core 2-way hyperthreaded Intel Xeon Platinum 8180 CPU 2.5GHz with 1.5TB of memory running Centos7 on Linux Kernel 3.10.0-862. This machine was part of the Odyssey Cluster and retrieves stored image data from Lustre mounted storage connected via 56 Gb/s FDR InfiniBand network.

LLSC Cluster is a shared supercomputing center LLSC (Lincoln Lab Supercomputing Center) [292] using the AMD Opteron partition of the TX-Green system, consisting of 274 compute nodes, each containing two 16-core AMD Opteron(TM) Processor 6274, running at 2.2GHz, for a total of 32 cores per node, with 128 GB per node. The nodes employ a shared Lustre filesystem and are connected with a 10 GigE Arista switch. A total of 170 Opteron compute nodes, and 64 Intel E5 nodes were available for our experiments on this platform. Experiments run on more than 4 nodes were run on LLSC's private cluster by a third party with guidance from the authors.

AWS Cluster is a 200-node cluster where each node is an 8-core version of the *Common Multicore* platform. AWS ParallelCluster software is used to manage the nodes as a SLURM cluster, and EFS is used as the cluster's shared filesystem.

6.8 Empirical analysis of Stacker's memory usage

This section provides details on the empirical memory usage of Stacker.

We ran an experiment to empirically measure the memory requirements of Stacker. We followed a similar methodology to that used in Section 6.3 to analyze Quilter's memory requirements. Regions of varied size were extracted from a section of the *human100* dataset, and then this section was duplicated to construct a stack of two identical sections. Stacker then executed to align this dataset. Figure 6-11 shows the results empirical experiments

constant	**value**
pixel resolution	3×3nm
tile height	2724 pixels
tile width	3128 pixels
raw tile	8.5 MB
JP2 tile	832 KB
tile metadata	4 KB
keypoint size	156 bytes
keypoints per image	8,000–20,000

region size	**# tiles**
1mm row	107
1cm row	1,066
10cm row	10,656
$1mm^2$ *section*	13,040
$1cm^2$ *section*	1,304,000
$10cm^2$ *section*	130,400,000

Figure 6-12: Values for the different constants needed to calculate memory usage of a 2D alignment algorithm

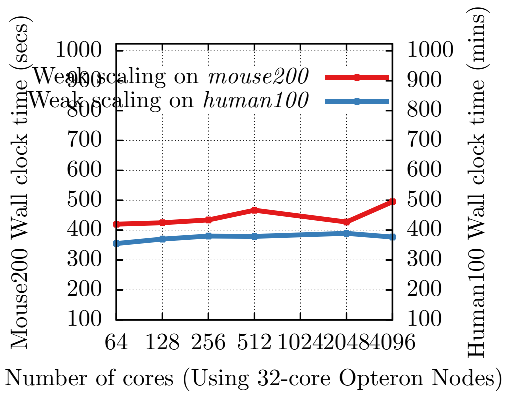
Dataset	Z	Size	Description
mouse200	200	2 TB	Subset of 100umSept2017 dataset from $100\mu^3$ volume of mouse brain, stored in J2K compressed format.
mouse50	50	0.55 TB	Subset of 100umIARPASep14 dataset from $100\mu^3$ volume of mouse brain, stored in JPEG compressed format.
human100	100	38 TB	ROI2w05 which is a subset of a 600 TB dataset obtained from human brain biopsy, stored in J2K compressed format.

Figure 6-13: Dataset descriptions. The Z column provides the number of $2D$ sections in the dataset. Datasets were obtained from the Lichtman Lab using the Zeiss multiSEM microscope.

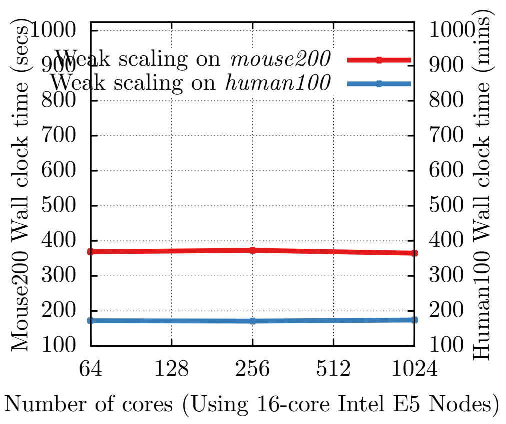
measuring Stacker's memory requirements. Stacker uses 300 KB of data per tile which is $\approx$ 20x more than analytic estimates. There are two reasons for the gap: (a) presently our implementation of Stacker uses 4-byte floats to represent keypoint descriptors where 1-byte is sufficient (since values are discretized into 255 bins); and, (b) aligning a section to its identical copy results in an abnormally large number of inter-section keypoint matches. As such, this experiment illustrates a worst-case scenario for Stacker's memory usage. Regardless, even in this scenario all datasets but the human brain remain a feasible task on commodity multicore hardware.

6.9 Conclusion

The alignment algorithm Quilter described in this chapter provides the first fast and accurate multicore alignment pipeline that can align data at $TB/$hr pace using commodity multicore hardware without compromising alignment accuracy.



(a) *LLSC Cluster* 32-core Opteron Nodes



(b) *LLSC Cluster* 16-core Intel E5 Nodes

Figure 6-14: Weak scalability experiments on the LLSC cluster. Illustrates reported runtimes on the *LLSC Cluster* platform on the *mouse200* and *human100* datasets when the stack input-size scales with the number of nodes used to execute Quilter and Stacker. The left y-axis provides the runtime in seconds for *mouse200* and the right y-axis provides the runtime in minutes for *human100*.

The design of Quilter was, to an extent, multicore-centric. Our development was principally performed using a single 18-core workstation and our main test dataset was the relatively small 550GB stack from *mouse50*. Through careful algorithmic design and careful consideration of how Quilter's memory requirements scale with dataset size, these algorithms easily scaled to datasets 100x larger.

Furthermore, the standard performance optimizations we employed to improve performance in the shared-memory setting was translatable to the distributed cluster-computing setting through coarse-grained parallelization across sections facilitated by Stacker's use of "associative" elastic transformations.

The performance of our alignment pipeline was achieved, in part, through aggressive downsampling of input images — an "optimization" that is often deemed unsuitable for connectomics due to its demand for highly accurate alignments. As we have illustrated, however, much of the performance benefits of downsampling can be achieved without sacrificing accuracy by using machine learning techniques to distinguish between highly-reliable and not-so-reliable results produced by a fast, but sometimes inaccurate, code path.

Chapter 7

Cilkmem: Algorithms for Analyzing the Memory High-Water Mark of Fork-Join Parallel Programs

This chapter describes the Cilkmem algorithms for analyzing the memory high-water mark of fork-join parallel programs. The Cilkmem algorithms, and their implementation in the Cilkmem tool, enable the identification of the worst-case memory requirements of shared-memory multicore codes. This work was conducted in collaboration with William Kuszmaul, Tao B. Schardl, and Daniele Vettorel.

Abstract

Software engineers designing recursive fork-join programs destined to run on massively parallel computing systems must be cognizant of how their program's memory requirements scale in a many-processor execution. Although tools exist for measuring memory usage during one particular execution of a parallel program, such tools cannot bound the worst-case memory usage over all possible parallel executions.

This chapter introduces Cilkmem, a tool that analyzes the execution of a deterministic Cilk program to determine its p-processor memory high-water mark (MHWM), which is the worst-case memory usage of the program over *all possible* p-processor executions. Cilkmem employs two new algorithms for computing the p-processor MHWM. The first algorithm calculates the exact p-processor MHWM in $O(T_1 \cdot p)$ time, where T_1 is the total work of the program. The second algorithm solves, in $O(T_1)$ time, the approximate threshold problem, which asks, for a given memory threshold M, whether the p-processor MHWM exceeds $M/2$ or whether it is guaranteed to be less than M. Both algorithms are memory efficient, requiring $O(p \cdot D)$ and $O(D)$ space, respectively, where D is the maximum call-stack depth of the program's execution on a single thread.

Our empirical studies show that Cilkmem generally exhibits low overheads. Across ten application benchmarks from the Cilkbench suite, the exact algorithm incurs a geometric-mean multiplicative overhead of 1.54 for $p = 128$, whereas the approximation-threshold algorithm incurs an overhead of 1.36 independent of p. In addition, we use Cilkmem to reveal and diagnose a previously unknown issue in a large image-alignment program contributing

to unexpectedly high memory usage under parallel executions.

7.1 Introduction

To design a recursive fork-join parallel program[1], such as a Cilk program, to run on massively parallel computing systems, software engineers must assess how their program's memory requirements scale in a many-processor execution. Many tools have been developed to observe a program execution and report its maximum memory consumption (e.g., [173, 307, 285, 276, 265]). But these tools can only ascertain the memory requirements of the one particular execution of the program that they observe. For parallel programs, whose memory requirements can depend on scheduling decisions that vary from run to run, existing tools are unable to provide bounds on the maximum amount of memory that might be used during future program executions[2]. This chapter studies the problem of computing the p-processor ***memory high-water mark*** (MHWM) of a parallel program, which measures the worst-case memory consumption of *any* p-processor execution. We introduce Cilkmem, an efficient dynamic-analysis tool that measures the MHWM of a Cilk program for an arbitrary number of processors p.

Computing the MHWM of an arbitrary parallel program is a theoretically difficult problem. In the special case where a program's allocated memory is freed immediately, without any intervening parallel control structure, computing the MHWM corresponds to finding a solution to the ***poset chain optimization problem*** [315, 59, 316]. The poset chain optimization problem is well understood theoretically, and the fastest known algorithms run in (substantial) polynomial time using techniques from linear programming [315]. A direct application of these algorithms to compute the MHWM of a parallel program would require computation that is polynomially large in the execution time of the original program.

Many dynamic-analysis tools (e.g., [105, 149, 303, 334, 351]) have been developed that exploit structural properties of fork-join programs to analyze a program efficiently. Specifically, these tools often leverage the fact that the execution of a fork-join program can be modeled as a series-parallel ***computation DAG*** (directed acyclic graph) [41, 105], where the edges model executed instructions, and the vertices model parallel-control dependencies.

But even when restricted to series-parallel DAGs, computing the p-processor MHWM efficiently is far from trivial. Identifying the worst-case memory requirement of a p-processor execution involves solving an optimization problem that sparsely assigns a finite number of processors to edges in the program's computation DAG. Such a computation DAG can be quite large, because of the liberal nature in which fork-join programs expose logically parallel operations. Moreover, whereas the poset chain optimization problem assumes that memory is freed immediately after being allocated, fork-join programs can free memory at any point that serially follows the allocation. Efficient solutions for this optimization problem are not obvious, and seemingly require a global view of the program's entire computation DAG. To obtain such a view, a tool would need to store a complete trace of the computation for offline processing and incur the consequent time and space overheads.

This work shows, however, that it is possible not only to compute the p-processor MHWM *efficiently* for a fork-join program, but also to do so in an *online* fashion, without

[1]When we talk about fork-join parallelism throughout this chapter, we mean recursive fork-join parallelism.

[2]In this chapter, when we consider executions of a program, we shall assume a fixed input to the program, including fixed seeds to any pseudorandom number generators the program might use.

```
MemoryExplosion($n$)
1  if $n > 1$
2      cilk_spawn MemoryExplosion($n - 1$)
3  $b \leftarrow$ malloc(1)
4  cilk_sync
5  free($b$)
6  return
```

Figure 7-1: Example Cilk program whose heap-memory usage can increase dramatically depending on how the program is scheduled.

needing to store the entire computation DAG. Specifically, we provide an online algorithm for computing the *exact* p-processor MHWM in $O(T_1 \cdot p)$ time, where T_1 is the total work of the program. We also examine the ***approximate threshold problem***, which asks, for a given memory threshold M, whether the p-processor MHWM exceeds $M/2$ or whether it is guaranteed to be less than M. We show how to solve the approximate threshold problem in $O(T_1)$ time using an online algorithm. Both of these algorithms are space efficient, requiring $O(p \cdot D)$ and $O(D)$ space, respectively, where D is the maximum call-stack depth of the program's execution on a single thread.

7.1.1 Memory consumption of fork-join programs

Let us review the fork-join parallel programming model and see how scheduling can cause a fork-join program's memory consumption to vary dramatically.

Recursive fork-join parallelism, as supported by parallel programming languages including dialects of Cilk [116, 219, 172], Fortress [3], Kokkos [101], Habanero [16], Habanero-Java [61], Hood [42], HotSLAW [258], Java Fork/Join Framework [208], OpenMP [275, 13], Task Parallel Library [218], Threading Building Blocks (TBB) [291], and X10 [68], has emerged as a popular parallel-programming model. In this model, subroutines can be spawned in parallel, generating a series-parallel computation DAG of fine-grained tasks. The synchronization of tasks is managed "under the covers" by the runtime system, which typically implements a randomized work-stealing scheduler [41, 116, 11, 38]. Constructs such as `parallel_for` can be implemented as syntactic sugar on top of the fork-join model. As long as the parallel program contains no determinacy races [105] (also called general races [266]), the program is ***deterministic***, meaning that every program execution on a given input performs the same set of operations, regardless of scheduling.

Even a simple fork-join program can exhibit dramatic and unintuitive changes in memory consumption,[3] based on how the program is scheduled on p processors. Consider, for example, the Cilk subroutine MemoryExplosion in Figure 7-1,[4] which supports parallel execution using the keywords `cilk_spawn` and `cilk_sync`. The `cilk_spawn` keyword on line 2 allows the recursive call to MemoryExplosion($n - 1$) to execute in parallel with the call to malloc(1) on line 3, which allocates 1 byte of heap memory. The `cilk_sync` on line 4 waits on the spawned recursive call to MemoryExplosion to return before proceeding; if a thread reaches the `cilk_sync`, and the recursive call to MemoryExplosion

[3]This work focuses on heap-memory consumption. In contrast, the Cilk runtime system is guaranteed to use stack space efficiently [41].

[4]Similar examples can be devised for other task-parallel programming frameworks.

has not yet completed, then the thread can be rescheduled to make progress elsewhere in the program.

Cilk's randomized work-stealing scheduler [41] schedules the parallel execution of MemoryExplosion as follows. When a Cilk worker thread encounters the `cilk_spawn` statement on line 2, it immediately executes the recursive call to MemoryExplosion$(n-1)$. If another worker thread in the system has no work to do, it becomes a ***thief*** and can ***steal*** the continuation of this parallel recursive call, on line 3.

Because of Cilk's scheduler, the memory consumption of MemoryExplosion can vary dramatically and nondeterministically from run to run, even though MemoryExplosion is deterministic. When run on a single processor, the `cilk_spawn` and `cilk_sync` statements effectively act as no-ops. Therefore, MemoryExplosion uses at most 1 byte of heap memory at any point in time, because each call to malloc is followed by a call to free almost immediately thereafter. When run on 2 processors, however, the memory consumption of MemoryExplosion can increase dramatically, depending on scheduling. While one worker is executing line 2, a thief can steal the execution line 3 and allocate 1 byte of memory before encountering the `cilk_sync` on line 4. The thief might then return to work stealing, only to find another execution of line 3 to steal, repeating the process. As a result, the heap-memory consumption of MemoryExplosion(n) on two or more Cilk workers can vary from run to run between 1 byte and n bytes, depending on scheduling happenstance.

7.1.2 Algorithms for memory high-water mark

This chapter presents algorithms for computing the p-processor MHWM of a program with a series-parallel computation DAG, and in particular, of a deterministic parallel Cilk program $\mathcal{P}$. Let $G = (V, E)$ be the computation DAG for $\mathcal{P}$, and suppose each edge of G is annotated with the allocations and frees within that edge.

Section 7.3 presents a simple offline algorithm for computing the exact p-processor MHWM of the parallel program $\mathcal{P}$, given the DAG G. A straightforward analysis of the exact algorithm would suggest that it runs in time $O(T_1 \cdot p^2)$, where T_1 is the 1-processor running time of the program $\mathcal{P}$. By performing an amortized analysis over the parallel strands of the program, we show that a slightly modified version of the algorithm actually achieves a running time of $O(T_1 \cdot p)$.

Explicitly storing the DAG G can be impractical for large programs $\mathcal{P}$. Section 7.4 presents a combinatorial restructuring of the exact algorithm that computes the MHWM in an *online* fashion, meaning that the algorithm runs as instrumentation on (a single-threaded execution of) the program $\mathcal{P}$. The online exact algorithm introduces at most $O(p)$ time and memory overheads when compared to a standard single-threaded execution of $\mathcal{P}$. In particular, the algorithm runs in time $O(T_1 \cdot p)$ and uses at most $O(p \cdot D)$ memory, where T_1 is the 1-processor running time of the program, and D is the maximum call-stack depth of the program's execution on a single thread. The simple amortization argument used for the offline algorithm does not apply to the more subtle structure of the online algorithm. Instead, we employ a more sophisticated amortized analysis, in which subportions of the graph are assigned sets of leader vertices, and the algorithm's work is charged to the leader vertices in such a way that no vertex receives more than $O(p)$ charge.

The two exact algorithms for computing the p-processor MHWM have the additional advantage that they actually compute each of the i-processor MHWM's for $i = 1, \ldots, p$. Thus a user can determine the largest $i \leq p$ for which the i-processor MHWM is below some threshold M.

We also consider the approximate-threshold version of the p-processor MHWM problem. Here, one is given a number of processors p and a memory threshold M, and wishes to determine whether p processors are at risk of coming close to running out of memory while executing on a system with memory M. Formally, an approximate-threshold algorithm returns a value of 1 or 0, where 1 indicates that the p-processor MHWM is at least $M/2$, and 0 indicates that the p-processor MHWM is bounded above by M.

Section 7.5 presents a strictly-linear time online algorithm for the approximate-threshold problem, running in time $O(T_1)$. The independence of the running time from p means that the algorithm can be used for an arbitrarily large number of processors p while still having a linear running time. This property can be useful for either understanding the limit properties of a program (i.e., behavior for very large p), or the behavior that a program will exhibit on a very large machine. The algorithm is also memory efficient. In particular, the memory usage of the algorithm never exceeds $O(D)$, where D is the maximum call-stack depth of the program's serial execution.

A key technical idea in the approximate-threshold algorithm is a lemma that relates the p-processor high-water mark to the *infinite-processor* MHWM taken over a restricted set of parallel execution states known as "robust antichains". The infinite-processor MHWM over robust antichains can then be computed in strictly linear time via a natural recursion. To obtain an online algorithm, we introduce the notion of a "stripped robust antichain" whose combinatorial properties can be exploited to remove dependencies between non-adjacent subproblems in the recursive algorithm.

7.1.3 The Cilkmem tool

We introduce the Cilkmem dynamic-analysis tool, which implements the online algorithms to measure the p-processor memory high-water mark of a deterministic parallel Cilk program.

Both of Cilkmem's algorithms run efficiently in practice. We implemented Cilkmem using the CSI framework for compiler instrumentation [302] embedded in the Tapir/LLVM compiler [304]. In Section 7.6, we measure the efficiency of Cilkmem on a suite of ten Cilk application benchmarks. Cilkmem introduces only a small overhead for most of the benchmarks. For example, the geometric-mean multiplicative overhead across the ten benchmarks is 1.54, to compute the MHWM exactly for $p = 128$, and 1.36, to run the approximate-threshold algorithm. For certain benchmarks with very fine-grained parallelism, however, the overhead can be substantially larger (although still bounded by the theoretical guarantees of the algorithms). We find that for these applications, the strictly-linear running time of the approximate-threshold algorithm provides substantial performance benefits, allowing computations to use arbitrarily large values of p with only small constant-factor overhead.

In addition to measuring Cilkmem's performance overhead, we use Cilkmem to analyze a big-data application, specifically, an image-alignment program [187] used for brain connectomics [247]. Section 7.6 describes how, for this application, Cilkmem reveals a previously unknown issue contributing to unexpectedly high memory usage under parallel executions.

7.1.4 Outline

The remainder of the chapter is structured as follows. Section 7.2 formalizes the problem of computing the p-processor MHWM in terms of antichains in series-parallel DAGs. Section 7.3 presents the $O(T_1 \cdot p)$-time exact algorithm, and Section 7.4 extends this to an online algorithm. Section 7.5 presents an online linear-time algorithm for the approximate-threshold problem. The design and analysis of the online approximate-threshold algorithm is the most technically sophisticated part of the chapter. Section 7.6 discusses the implementation Cilkmem, and evaluates its performance. Section 7.7 discusses related work, and Section 7.8 concludes with directions for future work.

7.2 Problem formalization

This section formalizes the problem of computing the p-processor memory high-water mark of a parallel program.

The DAG model of multithreading

Cilk programs express logical recursive fork-join parallelism through spawns and syncs. A ***spawn*** breaks a single thread into two threads of execution, one of which is logically a new child thread, while the other is logically the continuation of the original thread. A ***sync*** by a thread t, meanwhile, joins thread t with the completion of all threads spawned by t, meaning the next continuation of t occurs only after all of its current child threads have completed.

An execution of a Cilk program can be modeled as a computation DAG $G = (V, E)$. Each directed edge represents a ***strand***, that is, a sequence of executed instructions with no spawns or syncs. Each vertex represents a spawn or a sync.

The DAG G is a ***series-parallel DAG*** [105], which means that G has two distinguished vertices — a ***source*** vertex, from which one can reach every other vertex in G, and a ***sink*** vertex, which is reachable from every other vertex in G — and can be constructed by recursively combining pairs of series-parallel DAGs using series and parallel combinations. A ***series combination*** combines two DAGs G_1 and G_2 by identifying the sink vertex of G_1 with the source vertex of G_2. A ***parallel combination*** combines two DAGs G_1 and G_2 by identifying their source vertices with each other and their sink vertices with each other. We shall refer to any DAG used in a series or parallel combination during the recursive construction of G as a ***component*** of G. Although the recursive structure of series-parallel DAGs suggests a natural recursive framework for algorithms analyzing the DAG, Section 7.4 describes how a more complicated framework is needed to analyze series-parallel DAGs in an online fashion.

The structure of $G = (V, E)$ induces a poset on the edges E, in which $e_1 \leq e_2$ if there is a directed path from e_1 to e_2. A collection of edges $(e_1, e_2, \ldots, e_q)$ form an ***antichain*** if there is no pair e_i, e_j such that $e_i < e_j$. Note that edges form an antichain if and only if there is an execution of the corresponding parallel program in which those edges at some point run in parallel.

The p-processor memory high-water mark

To analyze potential memory usage, we model the computation's memory allocations and frees (deallocations) in the DAG G using two weights, $m(e)$ and $t(e)$, on each edge e. The weight $m(e)$, called the ***edge maximum***, denotes the high-water mark of memory usage at any point during the execution of e when only the allocations and frees within e are considered. The edge maximum $m(e)$ is always non-negative since, at the start of the execution of an edge e, no allocations or frees have been performed, and thus the (local) memory usage is zero. The weight $t(e)$, called the ***edge total***, denotes the sum of allocations minus frees over the entire execution of the edge. In contrast to $m(e)$, an edge total $t(e)$ can be negative when memory allocated previously in the program is freed within e.

The p-processor memory high-water mark is determined by the memory requirements of all antichains of length p or less in the computation DAG G. We define the ***water mark*** $W(A)$ of an antichain $A = (e_1, \ldots, e_q)$ to be maximum amount of memory that could be in use on a q-processor system that is executing the edges $e_1, \ldots, e_q$ concurrently. The ***p-processor high-water mark*** $H_p(G)$ is the maximum water mark over antichains of length p or smaller[5]:

$$H_p(G) = \max_{\substack{(e_1,\ldots,e_q)\in\mathcal{A},\\ q\le p}} W(e_1, \ldots, e_q), \tag{7.1}$$

where $\mathcal{A}$ is the set of antichains in G.

Memory water mark of an antichain

The water mark $W(A)$ of an antichain $A = (e_1, \ldots, e_q)$ is the sum of two quantities $W(A) = W_1(A) + W_2(A)$.

The quantity $W_1(A)$ consists of the contribution to the water mark from the edges $e_1, \ldots, e_q$ and from all the edges $e \in G$ satisfying $e \le e_i$ for some i:

$$W_1(A) = \sum_{e_i\in A} m(e_i) + \sum_{e\in E, e<e_i \text{ for some } e_i\in A} t(e). \tag{7.2}$$

The quantity $W_2(A)$ counts the contribution to the water mark of what we call ***suspended parallel components***. If the series-parallel construction of G combines two subgraphs G_1 and G_2 in parallel, we call them ***partnering parallel components*** of G. Consider two partnering parallel components G_1 and G_2, and suppose that G_2 contains at least one edge from the antichain A, while G_1 does not. Then there are two options for a parallel execution in which processors are active in the edges of A: either (1) the parallel component G_1 has not been executed at all, or (2) the parallel component G_1 has been executed to completion and is ***suspended*** until its partner parallel component completes. In the latter case, G_1 will contribute $\sum_{e\in G_1} t(e)$ to the water mark of A. If this sum, which is known as G_1's ***edge sum***, is positive, then we call G_1 a ***companion component*** to the antichain A. The quantity $W_2(A)$ counts the contribution to the water mark of edges in companion components. That is, if $\mathcal{G}$ is the set of companion components to A, then

$$W_2(A) = \sum_{G\in\mathcal{G}} \sum_{e\in G} t(e). \tag{7.3}$$

Note that the companion components of A are guaranteed to be disjoint, meaning that each edge total $t(e)$ in Equation (7.3) is counted at most once.

[5]This definition of water mark makes no assumption about the underlying scheduling algorithm. In particular, when a thread spawns, we make no assumptions as to which subsequent thread the scheduler will execute first.

The downset non-negativity property

Several of our algorithms, specifically for the approximate-threshold problem, take advantage of a natural combinatorial property satisfied by edge totals $t(e)$. Although $t(e)$ can be negative for a particular edge e, the sum $\sum_{e \in E} t(e)$ is presumed to be non-negative, since the parallel program should not, in total, free more memory than it allocates. We can generalize this property to subsets of edges, called downsets, where a subset $S \subseteq E$ is a ***downset*** if, for each edge $e \in S$, every edge $e' < e$ is also in S. The ***downset-non-negativity property*** requires that, for every downset $S \subseteq E$, $\sum_{e \in S} t(e) \geq 0$. This property corresponds to the real-world requirement that at no point during the execution of a parallel program can the total memory allocated be net negative.

7.3 An exact algorithm with $O(p)$ overhead

This section presents ExactOff, an $O(|E| \cdot p)$-time offline algorithm for exactly computing the high-water marks $H_1(G), \ldots, H_p(G)$ of a computation DAG G for all numbers of processors $1, \ldots, p$. We first give a simple dynamic-programming algorithm which runs in time $O(|E| \cdot p^2)$. We describe how ExactOff optimizes this simple algorithm. We then perform an amortization argument to prove that ExactOff achieves a running time of $O(|E| \cdot p)$.

The algorithm exploits the fact that G can be recursively constructed via series and parallel combinations, as Section 7.2 describes. The algorithm builds on top of this recursive structure. Note that one can construct a recursive decomposition of a series-parallel DAG G in linear time [335]. Section 7.4 discusses how to adapt the algorithm in order to run in an online fashion, executing along with the parallel program being analyzed, and introducing only $O(p)$ additional memory overhead.

Given a parallel program represented by a series-parallel DAG G, and a number of processors p, we define the $(p+1)$-element array $R_G = (R_G[0], R_G[1], \ldots, R_G[p])$ so that, for $i > 0$, $R_G[i]$ is the memory high-water mark for G over all antichains of size exactly i. We define $R_G[0]$ to be $\max(0, t(G))$, where $t(G) := \sum_{e \in G} t(e)$. For $i > 0$, if the graph G contains no i-edge antichains, then $R_G[i]$ is defined to take the special value null, treated as $-\infty$.

One can compute $H_p(G)$ from the array R_G using the identity $H_p(G) = \max_{i=1}^{p} R_G[i]$. Our goal is therefore to recursively compute R_G for the given DAG G.

An $O(|E| \cdot p^2)$-time algorithm

The algorithm computes R_G using the recursive series-parallel decomposition of G. When G consists of a single edge e, we have $R_G[0] = \max(0, t(e))$, $R_G[1] = m(e)$, and $R_G(2), \ldots, R_G[p] =$ null.

Suppose that G is the parallel combination of two graphs G_1 and G_2. Then,

$$R_G[i] = \begin{cases} \max(0, t(G)) & \text{if } i = 0, \\ \max_{j=0}^{i} R_{G_1}[j] + R_{G_2}[i-j] & \text{otherwise.} \end{cases}$$

In the second case, if either of $R_{G_1}[j]$ or $R_{G_1}[i-j]$ are null, then their sum is also defined to be null. Moreover, note that the definitions of $R_{G_1}[0]$ and $R_{G_2}[0]$ ensure that suspended components are treated correctly in the recursion.

Suppose, on the other hand, that G is the series combination of two graphs G_1 and G_2. Then R_G can be expressed in terms of R_{G_1}, R_{G_2}, $t(G_1)$, $t(G)$ using the equation,

$$R_G[i] = \begin{cases} \max(0, t(G)) & \text{if } i = 0, \\ \max(R_{G_1}[i], t(G_1) + R_{G_2}[i]) & \text{otherwise.} \end{cases}$$

Combining the above cases yields an $O(|E|{\cdot}p^2)$-time algorithm for computing R_G.

Achieving a running time of $O(|E|{\cdot}p)$

To optimize the simple algorithm, we define, for a DAG G, the value $s(G)$ to be the size of the largest antichain of edges in G, or p if G contains an antichain of size p or larger. The value $s(G)$ is easy to compute recursively using the recursion $s(G) = \min(s(G_1) + s(G_2), p)$, when G is the parallel combination of components G_1 and G_2, and $s(G) = \max(s(G_1), s(G_2))$, when G is the series combination of G_1 and G_2.

ExactOff optimizes the simple dynamic program as follows. Suppose that G is a parallel combination of components G_1 and G_2. Notice that $R_{G_1}[i] = \text{null}$ whenever $i > s(G_1)$ and $R_{G_2}[i] = \text{null}$ whenever $i > s(G_2)$. It follows that,

$$R_G[i] = \begin{cases} \max(0, t(G)) & \text{if } i = 0, \\ \displaystyle\max_{\substack{0 \le j \le i, \\ j \le s(G_1), \\ (i-j) \le s(G_2)}} R_{G_1}[j] + R_{G_2}[i-j] & \text{otherwise,} \end{cases} \tag{7.4}$$

where the max for the second case is defined to evaluate to null if it has zero terms.

Theorem 31 *For a series-parallel DAG $G = (V, E)$,* ExactOff *recursively computes R_G in time $O(|E|{\cdot}p)$.*

To prove Theorem 31, let us consider the time needed to compute R_G when G is obtained by combining two subgraphs G_1 and G_2 in parallel. For each value of $i \le s(G_1)$ and of $i - j \le s(G_2)$, the term $R_{G_1}[i] + R_{G_2}[i-j]$ will appear in Equation (7.4) for exactly one index i. It follows that the total time to compute R_G from R_{G_1} and R_{G_2} is at most $O(p + s(G_1) \cdot s(G_2))$.

Since parallel combinations cost $O(p + s(G_1) \cdot s(G_2))$ and series combinations cost $O(p)$, Theorem 31 reduces to,

$$\sum_{(G_1, G_2) \in \mathcal{C}} s(G_1) \cdot s(G_2) \le O(|E|{\cdot}p),$$

where the set $\mathcal{C}$ consists of all parallel combinations in the recursive construction of G.

Lemma 32 *Let $G = (V, E)$ be the series-parallel DAG modeling some parallel program's execution. Then,*

$$\sum_{(G_1, G_2) \in \mathcal{C}} s(G_1) \cdot s(G_2) \le O(|E|{\cdot}p),$$

where the set $\mathcal{C}$ consists of all parallel combinations in the recursive construction of G.

Proof. Call a parallel combination between two components G_1 and G_2 ***fully-formed*** if $s(G_1) = s(G_2) = p$. We claim that there are at most $O(|E|/p)$ fully-formed parallel combinations in the recursive construction of G. Consider the full recursive constructing G from individual edges using series and parallel combinations. Then each fully-formed parallel combination reduces the total number of components H satisfying $s(H) = p$ by one. On the other hand, the number of components satisfying $s(H) = p$ can only be increased when two components H_1, H_2 satisfying $s(H_1), s(H_2) < p$ are combined to form a new component H satisfying $s(H) = p$. The total number of such combinations is at most $|E|/p$, since each such H absorbs at least p edges. Since the number of components satisfying $s(H) = p$ is incremented at most $|E|/p$ times, it can also be decremented at most $|E|/p$ times, which limits the number of fully-formed parallel combinations to $|E|/p$.

Using the bound on the number of fully-formed parallel combinations, we have that

$$\sum_{(G_1,G_2)\in\mathcal{F}} s(G_1)\cdot s(G_2) \leq O\left(\frac{|E|}{p}\cdot p^2\right) \leq O(|E|{\cdot}p),$$

where $\mathcal{F}$ is the set of fully-formed parallel combinations.

To complete the proof of the lemma, it suffices to show

$$\sum_{(G_1,G_2)\in\overline{\mathcal{F}}} s(G_1)\cdot s(G_2) \leq O(|E|{\cdot}p), \tag{7.5}$$

where $\overline{\mathcal{F}}$ is the set of non-fully formed parallel combinations.

We prove Equation (7.5) with an amortization argument. Consider the recursive construction of G from edges via series and parallel combinations. Before beginning the combinations, we assign $2p-1$ credits to each edge $e \in E$. Every time two components G_1 and G_2 are combined in parallel and G_1 satisfies $s(G_1) < p$, we deduct $s(G_2)$ credits from each edge in G_1. Note that if both $s(G_1) < p$ and $s(G_2) < p$, then we deduct credits from the edges in both components.

The number of credits charged for each non-fully-formed parallel combination is at least $s(G_1)\cdot s(G_2)$. Thus the total number of credits deducted from all edges over the course of the construction of G is at least the left side of Equation (7.5). In order to prove Equation (7.5), it suffices to show that every edge still has a non-negative number of credits after the construction of G.

Consider an edge $e \in E$ as G is recursively constructed. Define H_t to be the component containing e after t steps in the construction, c_t to be the total amount of credit deducted from e in the first t steps, and a_t to be the size of the largest antichain in H_t. We claim as an invariant that $c_t \leq a_t$. Indeed, whenever $r = c_t - c_{t-1}$ credits are deducted from e during some step t, the parallel combination during that step also increases a_t to be at least r larger than a_{t-1}.

Since $s(H_t) = \min(a_t, p)$, the invariant tells us that whenever e is in a component H_t with $s(H_t) < p$, the total amount c_t deducted from e so far must satisfy $c_t < p$. Prior to the step t in which $s(H_t)$ finally becomes p, the total amount deducted from e is at most $p-1$. During the step t when $s(H_t)$ becomes p, at most p credits can be deducted from e. And after the step t when $s(H_t)$ becomes p, no more credits will ever be deducted from e. Thus the total deductions from e sum to at most $2p-1$, as desired. □

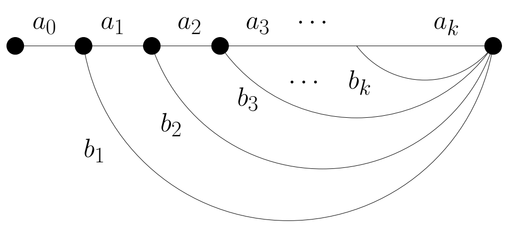


Figure 7-2: A multi-spawn combination. The components $a_0, \ldots, a_k$ and $b_1, \ldots, b_k$ are combined into a single component. If executed on a single processor in Cilk, the order of execution would be $a_0, b_1, a_1, b_2, a_2, \ldots, b_k, a_k$.

7.4 An online (memory-efficient) algorithm

The ExactOff algorithm in Section 7.3 computes $H_p(G)$ by considering the construction of a computation DAG G using only series and parallel combinations. Although in principle any series-parallel DAG can be constructed using only these combinations, doing so in an online fashion (as the parallel program executes) can require substantial memory overhead. In particular, parallel programs implemented in Cilk implicitly contain a third primitive way of combining components: multi-spawn combinations (see Figure Figure 7-2). A multi-spawn combination corresponds with all of the child spawns (i.e., `cilk_spawn` statements) of a thread that rejoin at a single synchronization point (i.e., at a `cilk_sync`).

When a multi-spawn combination is executed on a single processor, the execution traverses the components in the order $a_0, b_1, a_1, b_1, \ldots, b_k, a_k$. If one wishes to use the recursions from Section 7.3 in order to compute $H^{\circ}_{\infty}(G)$ then one must store the recursively computed values for each of $a_0, b_1, a_1, \ldots, a_k$ before any series or parallel combinations can be performed. After computing the values, one can then combine a_k and b_k in parallel, combine this with a_{k-1} in series, combine this with b_{k-1} in parallel, and so on.

When k is large, storing $\Theta(k)$ recursive values at a time can be impractical (though in total the memory overhead of ExactOff will still be bounded by the span of the parallel program). If one could instead design a recursion in which each multi-spawn combination could be performed using $O(p)$ space, then the recursive algorithm would be guaranteed to use no more than $O(p \cdot D)$ space, where D is the maximum stack depth of the parallel program in Cilk.

Appendix Section 7.9, presents the ExactOn algorithm, which implements this alternative recursion. The amortized analysis in Section 7.3 fails to carry over to ExactOn, because the work in the new algorithm can no longer be directly charged to the growth of components. Instead, we employ a more sophisticated amortized analysis in which components of the graph are assigned sets of leader vertices, and the work by the algorithm is charged to the leader vertices in such a way so that no vertex receives a charge of more than $O(p)$.

7.5 Online approximation in linear time

This section considers the approximate threshold version of the p-processor memory high-water mark problem. In particular, we construct a fully linear-time algorithm that processes a computation DAG $G = (V, E)$ and returns a boolean with the following guarantee: a return value of 0 guarantees that $H_p(G) \leq M$, while a return value of 1 guarantees that

$H_p(G) > M/2$.

The algorithm will compute the high-water mark over a special class of antichains that satisfy a certain property that we call ***stripped robustness***. Intuitively, the stripped robustness property requires that every edge e in the antichain contributes a substantial amount (at least $M/2p$) to the antichain's water mark. The algorithm solves the approximate threshold problem by computing the *infinite-processor* water mark over all stripped robust antichains, and then inferring from this information about $H_p(G)$.

Section 7.5.1 defines what it means for an antichain to be stripped robust and proves the correctness of the algorithm. Section 7.5.2 describes an online recursive algorithm for computing the quantity h needed by the algorithm in linear time $O(|E|)$. The algorithm uses space at most $O(D)$ where D is the maximum stack depth during an execution of the parallel program being analyzed. A simpler offline algorithm is given in Section 7.10.

7.5.1 Stripped robust antichains

This section defines a special class of antichains that we call stripped robust. We prove that, by analyzing stripped robust antichains with arbitrarily many processors, we can deduce information about $H_p(G)$.

An antichain A is ***stripped robust*** if it satisfies two requirements:

- **Large Local Contributions of Edges:** We define the ***local contribution*** $L^\bullet_A(x_i)$ of each edge $x_i \in A$ to the water mark $W(A)$ to be the value $W(A) - W(A \setminus \{x_i\})$. In order for A to be a stripped robust antichain, each x_i must satisfy $L^\bullet_A(x_i) > \frac{M}{2p}$.
- **Large Edge Contributions of Non-Critical Components:** For each multi-spawn combination $a_0, b_1, a_1, \ldots, a_k$ in G, if the component b_i contains at least one x_i, and if $a_i \cup b_{i+1} \cup \cdots \cup a_k$ contains at least one other x_j, then we call b_i a ***non-critical*** component. Define the ***local contribution*** of b_i to be $L^\bullet_A(b_i) = W(A) - W(A \setminus b_i)$, the reduction in water mark obtained by removing from A the edges also contained in b_i. In order for A to be a stripped robust antichain, each non-critical component b_i must satisfy $L^\bullet_A(b_i) > \frac{M}{2p}$.

The p-***processor robust memory high-water mark*** $H^\bullet_p(G)$ is defined to be

$$H^\bullet_p(G) = \max_{A \in \mathcal{S},\ |A| \le p} W(x_1, \ldots, x_q),$$

where $\mathcal{S}$ is the set of stripped robust antichains in E.

The first step in our approximate-threshold algorithm will be to compute the infinite-processor robust memory high-water mark $H^\bullet_\infty(G)$. Then, if $H^\bullet_\infty(G) \le M/2$, our algorithm returns 0, and if $H^\bullet_\infty(G) > M/2$, our algorithm returns 1.

Computing $H^\bullet_\infty(G)$ can be done online with constant overhead using a recursive algorithm described in Section 7.5.2. The computation is made significantly easier, in particular, by the fact that it is permitted to consider the infinite-processor case rather than restricting to p processors or fewer.

On the other hand, the fact that $H^\bullet_\infty(G)$ should tell us anything useful about $H_p(G)$ is non-obvious. In the rest of this section, we will prove the following theorem, which implies the correctness of the algorithm:

Theorem 33 *If $H^\bullet_\infty(G) \le M/2$, then $H_p(G) \le M$, and if $H^\bullet_\infty(G) > M/2$, then $H_p(G) > M/2$.*

It turns out that Theorem 33 remains true even if the second requirement for stripped robust antichains is removed (i.e. that non-critical components make large contributions). In fact, removing the second requirement (essentially) gives the notion of a ***robust antichain*** used in Section 7.10 in designing an *offline* algorithm for the same problem. As we shall see in Section 7.5.2, the second requirement results in several important structural properties of stripped robust antichains, making an *online* algorithm possible. The structural properties enable a recursive computation of $H^\bullet_\infty$ to handle multi-spawn combinations in a memory efficient fashion.

Our analysis begins by comparing $H^\bullet_p(G)$ to $H_p(G)$:

Lemma 34 $H^\bullet_p(G) \geq H_p(G) - \frac{M}{2}$.

Proof. Consider an antichain $A_1 = (x_1, \ldots, x_q)$, with $q \leq p$, that is not stripped-robust. We wish to construct a stripped robust antichain B satisfying $W(B) \geq W(A) - M/2$.

Then there must either be an edge $x_i \in A_1$ satisfying $L^\bullet_{A_1}(x_i) \leq \frac{M}{2p}$ or a non-critical component b_i satisfying $L^\bullet_{A_1}(b_i) \leq \frac{M}{2p}$. Define an antichain A_2 obtained by removing either the single edge x_i from A_1 (in the case where such an x_i exists) or all of the edges in $A_1 \cap b_i$ from A_1 (in the case where such a b_i exists). The antichain A_2 contains at least one fewer edges than does A_1, and satisfies $W(A_2) \geq W(A_1) - \frac{M}{2p}$.

If A_2 is still not stripped-robust, then we repeat the process to obtain an antichain A_3, and so on, until we obtain a stripped-robust antichain A_k. Because the empty antichain is stripped-robust, this process must succeed.

Since each antichain A_i in the sequence is smaller than the antichain A_{i-1}, the total number k of antichains in the sequence can be at most $q + 1$, where q is the size of the antichain $A_1 = (x_1, \ldots, x_q)$. On the other hand, since $W(A_i) \geq W(A_{i-1}) - \frac{M}{2p}$ for each $i \geq 2$, we also have that

$$W(A_k) \geq W(A_1) - (k-1) \cdot \frac{M}{2p} \geq W(A_1) - q \cdot \frac{M}{2p} \geq W(A_1) - \frac{M}{2},$$

as desired. □

Corollary 35 proves the first part of Theorem 33:

Corollary 35 *If $H^\bullet_\infty(G) \leq M/2 \leq M/2$, then $H_p(G) \leq M$.*

The second half of Theorem 33 is given by Lemma 36:

Lemma 36 *If $H^\bullet_\infty(G) > M/2$, then $H_p(G) > M/2$.*

Proof. Since $H^\bullet_\infty(G) > M/2$, there are two cases:

Case 1: There is a stripped robust antichain $A = (x_1, \ldots, x_q)$ with $q \leq p$ such that $W(A) > M/2$. In this case, we trivially get that $H_p(G) > M/2$.

Case 2: There is a stripped robust antichain $A = (x_1, \ldots, x_q)$ with $q > p$ such that $W(A) > M/2$. This case is somewhat more subtle, since the large number of edges in the antichain A could cause $W(A)$ to be much larger than $H_p(G)$. We will use the stripped robustness of A in order to prove that the potentially much smaller antichain $B = (x_1, \ldots, x_p)$ still has a large water mark $W(B) > \frac{M}{2}$.

Note that we cannot simply argue that $L^\bullet_B(x_i) \geq L^\bullet_A(x_i)$ for each $i \in \{1, \ldots, p\}$. In particular, the removal of edges from A may significantly change the local contributions of the remaining edges. Nonetheless, by exploiting the downset-non-negativity property we will still prove that $W(B) > \frac{M}{2}$.

For a given edge $x_i \in A$, define $\mathcal{T}_i$ to be the set of companion components T to the antichain A such that T is not a companion component to $A \setminus \{x_i\}$. Define P_i to be the set of edges e such that $e < x_i$ but $e \not< x_j$ for any other $x_j \in A$. Then the local contribution $L^{\bullet}_A(x_i)$ of x_i to A satisfies,

$$L^{\bullet}_A(x_i) \leq m(x_i) + \sum_{T \in \mathcal{T}_i} \sum_{e \in T} t(e) + \sum_{e \in P_i} t(e). \tag{7.6}$$

(This would be an exact equality if not for the fact that removing x_i from A can also introduce *new* companion components, which in turn reduces $L^{\bullet}_A(x_i)$.)

Let S_i denote the quantity on the right side of Equation (7.6). Since A is a stripped robust antichain, $S_i \geq M/2p$ for each i.

Now let us consider the water mark $W(B)$. For each $i \in \{1, \ldots, p\}$, each component $T \in \mathcal{T}_i$ is a companion component to B, just as it was to A. Moreover, each edge $e \in P_i$ continues to contribute $t(e)$ to the water mark of B. Define $\mathcal{T}$ to be the set of companion components to B that are not in any of $\mathcal{T}_1, \ldots, \mathcal{T}_p$, and P to be the set of edges e satisfying $e < x_i$ for some $i \in \{1, \ldots, p\}$ but $e \notin P_1 \cup \cdots \cup P_p$. Then the water mark $W(B)$ can be written as

$$\begin{aligned} W(B) &= \sum_{i=1}^{p} S_i + \sum_{T \in \mathcal{T}} \sum_{e \in T} t(e) + \sum_{e \in P} t(e) && (7.7) \\ &\geq M/2 + \sum_{T \in \mathcal{T}} \sum_{e \in T} t(e) + \sum_{e \in P} t(e). && (7.8) \end{aligned}$$

Since each $T \in \mathcal{T}$ satisfies $\sum_{e \in T} t(e) > 0$ (or else T would not be a companion component to B),

$$W(B) \geq M/2 + \sum_{e \in P} t(e).$$

In order to complete the proof that $W(B) > M/2$, it suffices by the downset-non-negativity property to show that P is a downset. Notice that P can be rewritten as

$$\begin{aligned} P &= \{e \in E \mid e < x_i \text{ for some } i = 1, \ldots, p\} \\ &\quad \cap \{e \in E \mid e < x_i \text{ and } e < x_j \text{ for some } x_i \neq x_j \in A\} \\ &= \left(\bigcup_{i=1}^{p} \{e < x_i\} \right) \cap \left(\bigcup_{x_i \neq x_j \in A} \{e < x_i\} \cap \{e < x_j\} \right). \end{aligned}$$

Since the downset property is closed under unions and intersections, it follows that P is a downset. □

7.5.2 Recursively computing $H^{\bullet}_{\infty}(G)$

This section discusses a recursive algorithm for computing $H^{\bullet}_{\infty}(G)$ in an online fashion. The algorithm treats G as being recursively constructed via series and multi-spawn combinations. For each multi-spawn combination, we assume we are recursively given the computed values for $a_0, b_1, a_1, \dots, a_k$, one after the other. Because k may be large, the recursion is not permitted to store these values. Instead it stores a constant amount of metadata that is updated over the course of the multi-spawn combination.

Finding a water-mark-maximizing stripped robust antichain A in a multi-spawn combination $C = (a_0, b_1, \dots, a_k)$ is complicated the following subtlety: if we choose to include an edge in one of the b_j's, then this may reduce the local contribution of any edges included in later a_j's and b_j's, resulting in those edges being unable to be included in the antichain. Therefore, greedily adding edges to the antichain A as we recursively execute $a_0, b_1, \dots, a_k$ may not result in an optimal stripped robust antichain.

The second requirement for stripped robust antichains (that non-critical components must make large edge contributions) is carefully designed to eliminate this problem. It allows us to prove the following lemma, which characterizes how non-critical components behave in water-mark-maximizing antichains A that contain multiple edges.[6]

Lemma 37 *Consider a multi-spawn combination C with components $a_0, b_1, a_1, \dots, a_k$. Consider $i \in \{1, \dots, k\}$, and suppose A is a stripped robust antichain in C that (1) contains multiple edges; (2) contains at least one edge in $a_i, b_{i+1}, \dots, a_k$; and (3) that achieves the maximum water mark over all stripped robust antichains in C that contain multiple edges.*

Let $t(b_i) = \sum_{e \in b_i} t(e)$, and let $m(b_i)$ denote the water mark of the best stripped robust antichain in b_i. (Note that $m(b_i)$ considers only the subgraph b_i.)

- *If $t(b_i) > 0$ and $m(b_i) \leq t(b_i) + \frac{M}{2p}$, then b_i is a companion component of A.*
- *If $t(b_i) \leq 0$ and $m(b_i) \leq \frac{M}{2p}$, then b_i is not a companion component of A and does not contribute any edges to A.*
- *If $m(b_i) > \max(0, t(b_i)) + \frac{M}{2p}$, then A restricted to b_i is a stripped robust antichain with water mark $m(b_i)$.*

Proof.

Any edges that b_i contributes to A must form a stripped robust antichain in b_i. The water mark s of that antichain within b_i can be at most $m(b_i)$. It follows that

$$L^{\bullet}_A(b_i) \leq m(b_i) - \max(0, t(b_i)),$$

since the removal of the edges in b_i from A will have the affect of (a) reducing the water mark by s and (b) introducing b_i as a companion component to A if $t(b_i) > 0$.

Since $m(b_i) - \max(0, t(b_i)) \leq \frac{M}{2p}$ in the first two cases of the lemma, b_i cannot contribute any edges to A in these cases. This ensures that in the first case b_i will be a companion component of A, and in the second case b_i will neither be a companion component nor contribute any edges.

The third case of the lemma is somewhat more subtle. Suppose that $m(b_i) > \max(0, t(b_i)) + \frac{M}{2p}$. We wish to show that A restricted to b_i is a stripped robust antichain with water mark $m(b_i)$. If A contains at least one edge in b_i, then since A has maximum water mark over multi-edge stripped robust antichains in C, it must be that A restricted to b_i is stripped robust and has water mark $m(b_i)$, as desired.

[6]In fact, the same lemma would be true if we removed the restriction that A contain multiple edges. The restriction is necessary for our applications of the lemma, however.

Suppose, on the other hand that A contains no edges in b_i. We will show that A does not achieve the maximum water mark over all stripped robust antichains in C that contain multiple edges. Define A' to be A with the addition of edges in b_i so that A' restricted to b_i is stripped robust and has water mark $m(b_i)$. Since $m(b_i) \geq \max(0, t(b_i)) + \frac{M}{2p}$, the water mark of A' must be more than $\frac{M}{2p}$ greater than that of A.

Since A has maximum water mark, and A' has a larger water mark, A' must no longer be stripped robust. Notice, however, that the local contribution of b_i in A' is greater than $\frac{M}{2p}$, and the local contributions of the other non-critical components of C in A' are the same as in A. Thus the only way that A' can no longer be stripped robust is if there is a single edge $x_j \in A'$ with local contribution at most $\frac{M}{2p}$. Moreover, x_j must be the only edge from A that is contained in any of the components $a_i, b_{i+1}, \ldots, a_k$. Define A'' to be A' with the edge x_j removed. Note that A'' has at least as many edges as did A initially, and is thus still a multi-edge antichain.

Since the local contribution $L^{\bullet}_{A'}(x_j)$ of x_j to A' was at most $\frac{M}{2p}$, the water mark of A'' still exceeds that of A. We claim, however, that A'' is a stripped robust antichain, contradicting that fact that A has maximum water mark out of all multi-edge stripped robust antichains. If b_i contains multiple edges in A'', then the fact that those edges form a stripped robust antichain when restricted to b_i, and that the other noncritical components and edges in A'' have the same local contributions to A'' as they did to A, ensure that A'' is a stripped robust antichain. If, on the other hand, b_i contains a single edge in A'', then the removal of that edge would reduce $W(A'')$ by at least as much as would have the removal of the edge x_j from the original antichain A (since both removals result in the same antichain). Thus $L^{\bullet}_{A''}(x_i) \geq L^{\bullet}_{A}(x_j) > \frac{M}{2p}$, ensuring that A'' is a stripped robust antichain. Since A'' has a larger water mark than A, we have reached a contradiction, completing the proof of the third case of the lemma. □

Our algorithm for computing $H^{\bullet}_{\infty}(G)$ recursively computes three quantities for each component C of the graph.

- The total allocation and freeing work done in C,

$$\text{MemTotal} = \sum_{e \in C} t(e).$$

- The memory high-water mark with one processor,

$$\text{MaxSingle} = H_1(C).$$

- The infinite-processor high-water mark restricted to stripped robust antichains containing *more* than one edge:

$$\text{MultiRobust} = \max_{A \in \mathcal{S},\ |A|>1} W(A),$$

  where $\mathcal{S}$ is the set of stripped robust antichains in C. If C contains no such multi-edge antichains, then MultiRobust = null.

The special handling in the recursion of antichains with only one edge (i.e., by MaxSingle) is necessary because the local contribution of that edge x is not yet completely determined until at least one other edge is added to the antichain. On the other hand, once a stripped robust antichain contains multiple edges, the local contribution of each edge is now fixed, even if we combine this antichain with other antichains as we recursively construct the graph. This allows for all multi-edge robust antichains to be grouped together in the variable MultiRobust.

As a base case, for a component C consisting of a single edge e, we initialize the variables as follows: MemTotal $= t(e)$, MaxSingle $= m(e)$, and MultiRobust = null.

When we combine two components C_1 and C_2 in series to build a new component C, we have,

$$C.\mathrm{MemTotal} = C_1.\mathrm{MemTotal} + C_2.\mathrm{MemTotal},$$

$$C.\mathrm{MaxSingle} = \max(C_1.\mathrm{MaxSingle}, C_1.\mathrm{MemTotal}_{C_1} + C_2.\mathrm{MaxSingle}),$$

$$C.\mathrm{MultiRobust} = \max(C_1.\mathrm{MultiRobust}, C_1.\mathrm{MemTotal} + C_2.\mathrm{MultiRobust}).$$

In the computations of $C.$MaxSingle and $C.$MultiRobust we implicitly use the fact that every antichain in C must either be in C_1 or in C_2. Moreover, the antichains in C_2 have water mark $C_1.$MemTotal greater in C than they did in C_2.

Note that the computation of $C.$MultiRobust would not be correct if MultiRobust were also considering single-edge antichains. In particular, the local contribution of an edge in a single-edge antichain A in C_2 will differ from the local contribution of the same edge in the same antichain in $C_1 \cup C_2$, allowing it to possibly form a stripped robust antichain in one but not the other. Because MultiRobust considers only multi-edge antichains, however, this is not a problem.

The recursion for combining components in a multi-spawn combination is more sophisticated. Consider a multi-spawn combination C as in Figure 7-2 with components $C_1 = a_0, C_2 = b_1, C_3 = a_1, \ldots, C_{2k+1} = a_k$. As our algorithm receives information on each of $C_1, C_2, C_3, \ldots$, it must gradually construct the best multi-edge stripped robust antichain in the multi-spawn component. Lemma 37 ensures that this is possible, because the role that each b_i plays in such an antichain depends only on whether any additional edges will be included from later components $a_i, b_{i+1}, \ldots$, and not on the specific properties of the components.

Nonetheless, the bookkeeping for the recursion is made subtle by the handling of suspended components and other casework. We defer the full recursion to Section 7.11.

7.6 Empirical evaluation

This section discusses the implementation and evaluation of Cilkmem on a suite of benchmark programs as well as on a large image processing pipeline performing image alignment.

7.6.1 Implementation

We implemented Cilkmem as a CSI tool [302] written in C++ for the Tapir compiler [304]. The following discussion describes how these facilities are used to implement Cilkmem's algorithms for MHWM analysis.

Benchmark	*Input size*	T_{exact}/T_1								
		p=32	p=64	p=128	p=256	p=512	p=1024	p=2048	p=4096	T_{approx}/T_1
strassen	4096 x 4096 matrix	1.00	1.00	1.00	0.98	1.02	0.98	0.98	1.00	0.99
nBody	1,000,000 points	1.00	1.00	1.00	0.99	0.99	0.99	1.00	1.00	1.00
lu	4096 x 4096 matrix	1.01	1.02	1.03	1.02	1.06	1.61	3.02	5.77	1.01
remDups	100,000,000 integers	1.03	1.03	1.03	1.01	1.02	1.04	1.58	3.05	1.02
dict	100,000,000 integers	1.11	1.10	1.12	1.11	1.10	1.12	1.66	3.17	1.10
ray	small	1.11	1.11	1.11	1.09	1.11	1.21	1.38	2.74	1.10
delaunay	5,000,000 points	1.16	1.16	1.17	1.13	1.13	1.15	1.16	1.19	1.16
nqueens	13 x 13 board	1.44	1.46	1.54	1.66	4.67	7.80	14.82	28.53	1.23
qsort	50,000,000 elements	3.63	3.73	4.07	5.18	12.75	28.60	53.89	103.63	2.61
cholesky	2000 x 2000 matrix	6.62	6.86	7.51	10.08	34.68	59.23	113.79	219.18	4.68

Table 7.1: Application benchmarks from the Cilkbench suite showing the overhead of Cilkmem over a single-threaded execution. The overhead is computed as the geometric-mean ratio of at least 5 runs with Cilkmem enabled and at least 5 runs of the un-instrumented program for each benchmark in the table.

The Cilkmem CSI tool tracks the evolution of a program's series-parallel DAG by inserting shadow computation before and after the instructions used by the Tapir compiler to represent fork-join parallelism. The language constructs used by the program-under-test to represent fork-join parallelism (e.g., Cilk's `cilk_spawn` and `cilk_sync` keywords) are lowered to Tapir's *detach* and *sync* instructions during compilation. The CSI framework provides instrumentation *hooks* that enable a tool to insert *shadow computation* before and after instructions in the compiler's intermediate representation of the program.

Memory allocations and frees are tracked by Cilkmem using process-wide hooks that intercept calls to the major allocation facilities provided by `glibc` via library interpositioning [54, Ch. 7.13]. These allocation facilities include `malloc`, `aligned_alloc`, `realloc` and `free`. While Cilkmem could use CSI's instrumentation hooks to track memory allocations, the use of interpositioning allows Cilkmem to capture calls to allocation functions performed in shared dynamic libraries that may not have been compiled with instrumentation enabled. Furthermore, interpositioning makes it possible for Cilkmem to track the requested sizes of allocations without the maintenance of an additional auxillary data structure by prepending to each allocation a small *payload* containing the size of the allocated block of memory. This payload is retrieved at deallocation time to determine how much memory has been freed. As an alternative to the payload-based technique, Cilkmem can also be instructed to retrieve the size of allocations using the Linux-specific function `malloc_usable_size`. The difference between the two methods comes down to whether the allocation size seen by Cilkmem is the *requested* size or the *usable* size determined by the memory allocator, which is allowed to reserve more memory than requested by the user. Special care is taken to ensure that the memory activity of Cilkmem's instrumentation is properly distinguished from activity originating from the program-under-test.

Cilkmem separates its instrumentation and analysis logic into two separate threads that communicate in a producer–consumer pattern. The producer thread executes the program-under-test and generates records that keep track of the allocation or deallocation of memory as well as the evolution of the series-parallel structure of the program execution. These records are sent to the consumer thread which executes either the exact or approximate MHWM algorithm in a manner that is fidelitous with the descriptions of the online MHWM

```
Memory high-water mark for p = 1 : 894727307 bytes
Source map for p = 1:
  [strassen.c:517]: 492013632 bytes
  [strassen.c:776]: 402653184 bytes
  [strassen.c:459]: 60491 bytes
Memory high-water mark for p = 2 : 1017641835 bytes
Source map for p = 2:
  [strassen.c:517]: 614928160 bytes
  [strassen.c:776]: 402653184 bytes
  [strassen.c:459]: 60491 bytes
Memory high-water mark for p = 3 : 1140556363 bytes
Source map for p = 3:
  [strassen.c:517]: 737842688 bytes
  [strassen.c:776]: 402653184 bytes
  [strassen.c:459]: 60491 bytes
```

Figure 7-3: An example of Cilkmem's verbose-mode output for the *strassen* benchmark. This excerpt shows the reported high-water mark in bytes for p from 1 to 3 and which lines of code contribute to it.

algorithms in Section 7.4 and Section 7.5. In order to maintain the theoretical space-bounds of the online MHWM algorithms, the Cilkmem tool's producer thread blocks in the rare case there is a backlog of unconsumed records. Although it would be possible to implement the online MHWM algorithms without the use of this producer–consumer pattern, such an approach can result in increased instrumentation overhead due to, among other things, an increase in instruction cache misses.

In addition to computing the memory high-water mark, Cilkmem supports a *verbose* mode which provides the user with actionable information for identifying the root cause of the memory high-water mark. In verbose mode, Cilkmem constructs the full computation DAG, annotates each edge (strand) with information about how much memory is allocated within that strand, and outputs the resulting graph in a graphical format. Furthermore, Cilkmem identifies the lines of code responsible for allocations that contribute to the memory high-water mark for a given p, and reports them to the user along with how many bytes each line is responsible for.

Figure Figure 7-3 shows the output genenerated by Cilkmem's verbose mode when run on the *strassen* benchmark from the Cilkbench suite with a 4096×4096 matrix as the input. The program's memory high-water mark increases by about 122MB for each additional processor, and Cilkmem identifies line 517 of source file `strassen.c` as the responsible for such increase. Cilkmem also reveals that the allocations performed by two other lines of code do not increase in size as p increases.

7.6.2 Benchmarks

We tested the runtime overhead of Cilkmem on ten Cilk programs from the Cilkbench suite[7]. The Cilkbench suite contains a variety of programs that implement different kind of algorithms such as Cholesky decomposition, matrix multiplication, integer sorting, and Delaunay triangulation.

Table Table 7.1 shows the geometric-mean overhead as the ratio between an execution of the benchmark program through Cilkmem and a serial execution of the uninstrumented

[7]The Cilkbench suite is at https://github.com/neboat/cilkbench

```
Differential MHWM for p = 3 -> p = 4
  [sift.cpp:355]:    63.79 MB
  [sift.cpp:259]:    21.26 MB
  [sift.cpp:319]:     7.08 MB
  [tile.cpp:2039]:    5.31 MB
  [sift.cpp:327]:   226.76 MB
```

Figure 7-4: Differential MHWM report on image alignment application. Illustrates the output of Cilkmem's differential MHWM report showing the relative increase in the MHWM when increasing the number of processors from $p = 3$ to $p = 4$.

program (T_1). The performance of Cilkmem was tested for both the exact algorithm and for the approximate-threshold algorithm. When the exact algorithm was used, Cilkmem was run with p set to all powers of 2 from 32 to 4096 inclusive and its runtime is reported in the table as T_{exact}. The approximate-threshold algorithms's runtime (T_{approx}) is independent of p.

As can be evinced from the results, Cilkmem's overheads are generally low and typically result in less than a 20% overhead relative to an uninstrumented execution. Cilkmem's overheads are especially low for benchmarks that do not exhibit substantial fine-grained parallelism. For programs that do (e.g., `qsort` or `cholesky`), however, the exact MHWM algorithm can incur a significant performance degradation for large values of p. In these cases, the approximate-threshold algorithm is *substantially* faster than the exact algorithm.

7.6.3 Optimizations

The Cilkmem tool implements two critical optimizations to achieve low instrumentation overheads.

Many Cilk programs do not exhibit memory activity in every strand. The MHWM algorithms were optimized to avoid performing unecessary work whenever a component is guaranteed not to contribute to the water mark. This often allows Cilkmem to quickly skip over large sections of the series-parallel structure.

As outlined in section Section 7.6.1, Cilkmem utilizes two threads which act as a producer and as a consumer. Since the data produced (allocated) by the first thread is consumed (freed) by the second in FIFO order, the memory management of Cilkmem's internal data structures can be greatly simplified to avoid a large number of small allocations and deallocations. Memory is managed in a memory pool that takes advantage of the FIFO nature of the producer-consumer relation.

7.6.4 Case study: multicore image processing pipeline

We conducted a case study on an existing image processing pipeline [187] that performs alignment and reconstruction of high-resolution images produced via electron microscopy[8]. The alignment code processes thousands of 8.5 MB image tiles in order to stitch them together to form a 2D mosaic. The pipeline was designed to carefully manage memory resources so as to be able to 2D align very large mosaics on a single multicore. The memory usage of the application scaled predictably as a function of the size of the mosaic, but there was an unexplained increase in the memory usage when adding additional processors. Given the size of the individual images being aligned, a natural expectation would be for the MHWM to increase by approximately 8.5 MB per processor. Empirically, however,

[8]For the purposes of this study, we ran the pipeline of [187] on full-resolution images without using FSJ.

the application's 18-core execution used several gigabytes more memory than the 1-core execution, and the precise amount of extra memory used varied from run-to-run.

We used Cilkmem to analyze the pipeline's p-processor MHWM. Cilkmem revealed that the MHWM increased by approximately 350 MB per processor. To identify the source of this per-processor memory requirement, we used Cilkmem to generate a differential MHWM report[9] that attributes an increase in the MHWM between p and $p + 1$ processors to particular source-code locations.

Figure 7-4 shows the differential MHWM report generated by Cilkmem for the alignment code, which reveals that lines 259, 327, and 355 of `sift.cpp` are responsible for increasing the MHWM of the application by a total of approximately 311 MB per processor. These lines perform allocations to store a difference-of-gaussian pyramid of images as a part of the scale-invariant feature transform employed by the alignment code. This procedure doubles the resolution of the images (to allow for subpixel localization of features), performs 6 GaussianBlurs on the images, downsamples the blurred image by a factor of 2 in each dimension, and repeats until the image size falls below a threshold. This routine is called on overlapping regions of image tiles. Although these overlapping regions are typically small, the largest 1% are as large as 2 MB in size. Whereas the original input images represent each pixel with a single byte, this procedure generates intermediate results that use 4-byte floating-point values for each pixel. A back-of-the-envelope calculation revealed that this procedure can require 200–400MB of space for tile pairs with large overlapping regions, which conforms with Cilkmem's analysis.

This study illustrates a case in which Cilkmem allowed application developers to make precise their memory-use projections for their pipeline, and illuminated a source of memory blow-up when running the pipeline on very-large multicores. Cilkmem's low instrumentation overhead made it practical to perform frequent and iterative tests on multiple versions of the pipeline. In fact, both the exact and approximate Cilkmem MHWM algorithms had less than 5% overhead on the alignment application for $p = 128$. These low overheads, coupled with the illuminating insights provided by Cilkmem, led to the incorporation of Cilkmem into the regression testing process for the alignment pipeline.

7.7 Related work

This section overviews related work in analysis of parallel programs, focusing on analysis of memory consumption and parallel-program analyses that do not depend on the particulars of the task-parallel program's scheduling.

Related theoretical work

From a theoretical perspective, the memory high-water mark problem (MHWM) is closely related to the ***poset chain optimization problem (PCOP)*** [315, 59, 316, 243]. In an instance of PCOP, one is given a parameter p and an arbitrary DAG $G = (V, E)$ in which each edge has been assigned a non-negative weight, and one wishes to determine the weight of the heaviest antichain containing p or fewer edges (where the weight of each antichain is the sum of the weights of its edges). Shum and Trotter [315] showed that PCOP can be solved in (substantial) polynomial time in the size of G using a linear-programming based

[9]Cilkmem generates differential MHWM reports via a lightweight script that parses Cilkmem's verbose-mode output.

algorithm; for the special case of $p = \infty$, an algorithm based on maximum-flows is also known [59, 316, 243].

The relationship between PCOP and MHWM was previously made explicit by Marchal *et al.* [243], who applied algorithms for PCOP in order to design polynomial-time algorithms for computing the memory high-water mark of parallel algorithms. Because none of the known algorithms for PCOP run in (even close) to linear time, however, the memory high-water mark algorithms of [243] are too inefficient to be used in practice. Additionally, in order to apply PCOP algorithms to the computation of memory high-water marks, [243] were forced to make a number of simplifying assumptions about the parallel programs being analyzed, and their algorithms require that the parallel programs be in what they call the simple-data-flow model.

The difficulty of solving PCOP efficiently on arbitrary DAGs motivates the focus in this chapter on the important special case in which the DAG G is series-parallel. Moreover, by modeling memory with arbitrary allocations and frees (rather than using the simple-data-flow model) we ensure that our algorithms have theoretical guarantees when applied to analyzing arbitrary task-parallel programs.

In addition to considering the high-water mark problem, Marchal *et al.* [243] considered the problem of adding new dependencies to a parallel program's DAG in order to reduce the high-water mark. They prove that this problem is NP-complete, and empirically evaluate several heuristics. These techniques would be difficult to apply to most real-world parallel programs, however, since they require the offline analysis of the parallel program's computation DAG.

Related tools

In practice, many tools exist to measure and report on the maximum memory consumption of a running program. For example, the Linux kernel tracks memory-usage information for every running process and publishes that information through the `proc` pseudo-filesystem [285], including the virtual memory and ***resident set size*** (RSS), which is the portion of the process's memory stored in main memory and, therefore, is upper-bounded by the memory high-water mark. Performance-analysis toolkits including Intel VTune Amplifier [173], the Sun Performance Analyzer [276], the Massif tool [336] in the Valgrind tool suite [265], and Linux's `memusage` tool [307] measure the memory consumption of an execution of a specified program and reports its peak stack- and heap-memory consumption. Like Cilkmem, these tools use intercept system calls to dynamic memory-allocation functions, such as `malloc` and `free`. Unlike Cilkmem, however, all of these memory-analysis tools gather information that is specific to how the program was scheduled for a particular run. These analyses do not analyze the worst-case memory consumption of any parallel execution of the program on a given processor count.

Several other dynamic-analysis tools for task-parallel programs have been developed whose analyses do not depend on the scheduling of the program. Tools such as Cilkview [149] and Cilkprof [303] analyze the execution of a Cilk program and report on the program's parallel scalability, which reflects how much speedup the program might achieve using different numbers of parallel processors. Several other tools analyze parallel memory accesses in a task-parallel program that might exhibit nondeterministic behavior between runs of the program [105, 20, 334, 287, 288, 97]. Like Cilkmem, the analyses performed by these tools do not depend on how the program was scheduled for a particular run, and instead provide insight into the behavior or performance of all parallel executions of the program. Unlike Cilkmem, however, these tools do not analyze memory consumption.

7.8 Conclusion

This chapter introduced Cilkmem, a tool that analyzes the p-processor memory high-water mark of fork-join programs. Cilkmem is built on top of novel algorithms which provide Cilkmem with both accuracy and running-time guarantees. We conclude with several directions of future work.

Although Cilkmem analyzes the behavior of parallel programs, currently Cilkmem is forced to run these programs in serial while performing the analysis. Extending Cilkmem to run in parallel is an important direction of future work.

Theoretically, all of our algorithms could be implemented in parallel at the cost of requiring additional memory. In particular, both online algorithms adapt to this setting using total space $O(p \cdot T_\infty)$ for the exact algorithm and $O(T_\infty)$ for the approximate-threshold algorithm, where T_∞ is the span of the parallel program being analyzed. This can be significant, especially for parallel programs with large multi-spawn combinations. Furthermore, there are technical challenges in parallelizing Cilkmem. The current instrumentation approach is not easily amenable to parallelization since thread scheduling is hidden by the Cilk runtime system. Finally, capturing memory allocations in a multi-threaded program is made more difficult by the fact that each allocation needs to be properly attributed to the correct thread and strand.

A theoretically interesting direction of work would be to extend our work on approximation algorithms to consider the memory-high water mark on parallel programs with *arbitrary* DAGs. Whereas computing the exact memory-high water mark of an arbitrary DAG is known to be difficult to do with low overhead, much less theoretical work has been done on the approximation version of the same question.

7.9 Appendix: Online exact computation of $H_p(G)$

This section describes ExactOn, an online algorithm to compute the exact memory high-water mark for processor counts $1, \ldots, p$. ExactOn adapts the $O(|E| \cdot p)$-time exact algorithm, ExactOff, from Section 7.3 to space-efficiently handle multi-spawn combinations.

Formally, ExactOn recursively computes three quantities for each component C: (1) The $(p+1)$-element array R_c; (2) the total memory allocated $t(C)$ over the edges in C; and (3) the value of $s(C)$. Since $s(C)$ can be recovered in time $O(p)$ from R_C, the final of these quantities can be computed non-recursively for each component.

Now let us consider a multi-spawn combination $C = (a_0, b_1, a_1, \ldots, b_k, a_k)$, and let $x_1, x_2, \ldots, x_{2k+1}$ be a consecutive labeling of $a_0, b_1, a_2, \ldots, b_k$ (i.e., $x_1 = a_0, x_2 = b_1, \ldots$). During a multi-spawn combination, we are given the values of $R_{x_i}, t(x_i), s(x_i)$ for each $i = 1, \ldots, 2k+1$, and we wish to compute R_C and $t(C)$ (after which we can obtain $s(C)$ from R_C in time $O(p)$).

The quantity $t(C)$ is easy to recursively compute, since $t(C) = \sum_i t(x_i)$. What's more difficult is to obtain the array R_C. To do this, as we receive $R_{x_l}, t(x_l), s(x_l)$ for each l, we maintain three intermediate variables. In order to define our intermediate variables, we must first introduce the notions of suspended-end and ignored-end water marks of antichains in a multi-spawn component C.

If an antichain A in C contains only edges in $b_1, \ldots, b_k$, and b_t is the largest t such that b_t contains a edge in A, then we say A has a ***suspended end*** if the components $a_{t+1}, b_{t+2}, \ldots, a_k$ form a companion component of A (which occurs if the sum of their edge costs is net positive). The ***suspended-end water mark*** of A is $W(A)$ if A has a suspended end, and is $W(A) + \sum_{e \in \bigcup a_{t+1}, b_{t+2}, \ldots, a_k} t(e)$ if A does not have a suspended end (i.e., it is the water mark A would have if it had a suspended end). Similarly, the ***ignored-end water mark*** of A is $W(A)$ if A does not have a suspended end, and is $W(A) - \sum_{e \in \bigcup a_{t+1}, b_{t+2}, \ldots, a_k} t(e)$ if A does have a suspended end. Additionally, for any antichain A that contains an edge in some a_t, we define the ***ignored-end water mark*** of A to be the water mark of A; thus the ignored-end water mark is defined for all antichains in A of C.

As we receive $R_{x_l}, t(x_l), s(x_l)$ for each l, we maintain three intermediate variables, each of which is an $O(p)$-element array indexed either by $i = 0, \ldots, p$ or $i = 1, \ldots, p$. Define l_1 to be the index of the largest-indexed a_i before or at x_l, and l_2 to be the index of the largest-indexed b_i before or at x_l. After receiving $R_{x_l}, t(x_l), s(x_l)$, the l-th entry of each of our three intermediate variables are updated to be defined as follows:

- $\text{SuspendEnd}_l[i]$ for $i = 1, \ldots, p$: This is the maximum suspended-end cost of any antichain A in $x_1 \cup \cdots \cup x_l$ such that (1) A contains exactly i edges; (2) all of A's edges are in $b_1, \ldots, b_{l_2}$. (Note that here $x_1 \cup \cdots \cup x_l$ is treated as a multi-spawn component and the costs of the antichains are considered just within the graph containing $x_1, \ldots, x_l$.)

- $\text{IgnoreEnd}_l[i]$ for $i = 1, \ldots, p$: This is the maximum ignored-end cost of any antichain A in $x_1 \cup \cdots \cup x_l$ that contains exactly i edges. If no such A exists, this is null, and is treated as $-\infty$.

- $\text{Partial}_l[i]$ for $i = 0, \ldots, p$: Consider antichains $A_1, \ldots, A_{l_2}$ in $b_1, \ldots, b_{l_2}$, respectively, such that the total number of edges in the antichains is i. Then $\text{Partial}_l[i]$ is the sum of two quantities: (1) The sum of the edge-totals in the a_i's seen so far, given by $\sum_{i=0}^{l_1} t(a_i)$; and the maximum possible value of the sum $\sum_{j=1}^{l_2} W(A_i)$, where the water mark of each A_i is considered within only the graph b_i, and the water mark of an antichain A_i containing no edges is set to $\max(0, t(b_i))$. (If no such $A_1, \ldots, A_{l_2}$ exist then $\text{Partial}_l[i] = \text{null}$.)

  In terms of $R_{b_j}[t_i]$, one can express $\text{Partial}_l[i]$ as

$$\text{Partial}_l[i] = \sum_{i=0}^{l_1} t(a_i) + \max_{t_1 + t_2 + \cdots + t_{l_2} = i} \sum_{j=1}^{l_2} R_{b_j}[t_j]. \tag{7.9}$$

When $l = 0$, before starting the computation, we initialize each entry of each of the intermediate variables to null; the exception to this is $\text{Partial}_0[0]$ which we initialize to zero.

Upon receiving a given x_l for an odd l (meaning $x_l = a_{l_1}$), we can compute the new intermediate variables as follows:

- $\text{SuspendEnd}_l[i]$ **for** $i = 1, \ldots, p$**:** This equals $\text{SuspendEnd}_{l-1}[i] + t(x_l)$.

- $\text{IgnoreEnd}_l[i]$ **for** $i = 1, \ldots, p$**:** This equals

$$\max\left(\text{IgnoreEnd}_{l-1}[i], \max_{j=1}^{i} \text{Partial}_{l-1}[i-j] + R_{x_l}[j]\right).$$

In particular, the right side of the maximum considers as an option for $\text{IgnoreEnd}_l[i]$ the possibility that our antichain A has some non-zero number j of edges in a_{l_1}, and then $i - j$ edges spread across $b_1, \ldots, b_{l_2}$.

- $\text{Partial}_l[i]$ **for** $i = 0, \ldots, p$**:** This is $\text{Partial}_{l-1}[i] + t(x_l)$.

Upon receiving a given x_l for an even l (meaning $x_l = b_{l_2}$), we can compute the new intermediate variables as follows:

- $\text{SuspendEnd}_l[i]$ **for** $i = 1, \ldots, p$**:** This equals

$$\max\left(\text{SuspendEnd}_{l-1}[i] + t(b_{l_2}), \max_{j=1}^{i} \text{Partial}_{l-1}[i-j] + R_{x_l}[j]\right).$$

In particular, the right side of the maximum considers as an option for $\text{SuspendEnd}_l[i]$ the possibility that our antichain A has some non-zero number j of edges in b_{l_2}, and then $i - j$ edges spread across $b_1, \ldots, b_{l_2-1}$. Note that the suspended-end cost and ignored-end cost of such an antichain in $x_1 \cup \cdots \cup x_l$ will be equal (since there is no end to be suspended); consequently, we will use a similar maximum to compute the new value of $\text{IgnoreEnd}_l[i]$.

- $\text{IgnoreEnd}_l[i]$ **for** $i = 1, \ldots, p$**:** This equals

$$\max\left(\text{IgnoreEnd}_{l-1}[i], \max_{j=1}^{i} \text{Partial}_{l-1}[i-j] + R_{x_l}[j]\right).$$

As before, the right side of the maximum considers as an option for $\text{IgnoreEnd}_l[i]$ the possibility that our antichain A has some non-zero number j of edges in b_{l_2}, and then $i - j$ spread across $b_1, \ldots, b_{l_2-1}$.

- $\text{Partial}_l[i]$ **for** $i = 0, \ldots, p$**:** This equals

$$\max\left(\text{Partial}_{l-1}[i] + R_{x_l}[0], \max_{j=1}^{i} \text{Partial}_{l-1}[i-j] + R_{x_l}[j]\right),$$

where the right side of the maximum is null for $i = 0$.

Once again, the right side of the maximum considers as an option for $\text{IgnoreEnd}_l[i]$ the possibility that our antichain A has some non-zero number j of edges in b_{l_2}, and then $i - j$ spread across $b_1, \ldots, b_{l_2-1}$. The left side, on the other hand, represents the case where no antichain edges appear in b_{l_2}.

Using the recursions above, we can compute the intermediate values for each l in time $O(p^2)$. We can then compute R_C using the identity

$$R_C[i] = \max(\text{SuspendEnd}_{2k+1}[i], \text{IgnoreEnd}_{2k+1}[i]),$$

for $i > 0$ and $R_C[0] = \max(0, t(C))$.

For a series-parallel graph G, we now have a online (space-efficient) algorithm for computing R_G in time $O(|E| \cdot p^2)$. Using a similar optimization as in the previous section, we can improve this running time to $O(|E| \cdot p)$. In particular, for each Partial_l, we keep track of the largest index containing a non-null entry; when computing each of the maximums in our updates, we can then ignore any terms involving a null entry of either Partial_{l-1} or of R_{x_l}. This ensures that computing the intermediate variables for a given value of l takes time at most

$$O\left(p + \min\left(\sum_{j=1}^{\lfloor (l-1)/2 \rfloor} s(b_j), p\right) \cdot s(x_l)\right). \tag{7.10}$$

Call this the ***optimized algorithm***. Using an amortized analysis we will prove that the optimized algorithm has running time $O(|E|{\cdot}p)$.

Theorem 38 *For a graph $G = (V, E)$ recursively constructed with series and multi-spawn combinations, the optimized algorithm recursively computes R_G in time $O(|E|{\cdot}p)$ and using space $O(p)$ for each combination.*

The fact that the product in Equation (7.10) involves a summation $\sum_{j=1}^{\lfloor (l-1)/2 \rfloor} s(b_j)$ means that the simple credit-charging argument used to prove Theorem 31 no longer suffices for proving Theorem 38. Nonetheless, by splitting the problem into two separate amortization arguments we are able to complete the analysis. This is done in Lemma 39.

Lemma 39 *Consider a series-parallel graph $G = (V, E)$ recursively built from series and multi-spawn parallel combinations. Denote each such multi-spawn combination by the tuple $(x_1, x_2, x_3, x_3, \ldots, x_t)$, where the odd-indexed x_i's represent the a_i components and the even-indexed x_i's represent the b_i components. Then*

$$\sum_{(x_1,\ldots,x_t)\in\mathcal{M}} \sum_{i=1}^{t-1} \min\left(\sum_{j=1}^{\lfloor i/2 \rfloor} s(x_{2j}), p\right) \cdot s(x_i) \le O(|E|{\cdot}p), \tag{7.11}$$

where the set $\mathcal{M}$ contains all multi-spawn combinations in the recursive construction of G.

Proof.

We think of each multi-spawn combination $(x_1, \ldots, x_t)$ as consisting of $t-1$ sub-combinations, in which after i sub-combinations we have combined $x_1, \ldots, x_{i+1}$. One should think of the cost of the $(i-1)$-th sub-combination is

$$\min\left(\sum_{j=1}^{\lfloor i/2 \rfloor} s(x_{2j}), p\right) \cdot s(x_i).$$

We say the sub-combination is ***heavy*** if $s(x_i) = p$ and is ***light*** if $s(x_i) < p$.

The light sub-combinations can be handled using a similar argument as for the non-fully-formed case in Lemma 32. At the beginning of the recursive construction of G, assign to each edge $2p-1$ credits. For each light sub-combination, combining some $x_1, \ldots, x_{i-1}$ with some x_i, we deduct $\min\left(\sum_{j=1}^{\lfloor i/2 \rfloor} s(x_{2j}), p\right)$ credits from each edge in x_i. Since the number of edges in x_i is at least $s(x_i)$, the sub-combination deducts a total of at least $\min\left(\sum_{j=1}^{\lfloor i/2 \rfloor} s(x_{2j}), p\right) \cdot s(x_i)$ credits. In order to bound the contribution of light sub-combinations to Equation (7.11), it suffices to prove that each edge $e \in E$ has a total of at most $2p-1$ credits deducted from it. This follows by the exact same invariant-based argument as used in the proof of Lemma 32.

In order to analyze the contribution of the heavy sub-combinations to Equation (7.11), we introduce a second amortization argument. Again we assign credits to edges, this time giving p credits to each edge e. As we recursively construct G through series and multi-spawn combinations, we assign to each component C a set of up to p ***representative edges***, which includes all of C's edges when C contains p or fewer edges, and p edges otherwise. When a component C is constructed by combining two components C_1 and C_2 in series, C's representative edges are the union of C_1's and C_2's (truncated to at most p edges). When a component C is constructed by a multi-spawn combination $(x_1, \ldots, x_t)$ such that at least one of the x_i's contains p or more edges, C's representative edges are inherited from the first such x_i; if none of the x_i's contain p or more edges, C's representative edges are the union of the representative edges for each of x_i's (truncated to at most p edges).

Now consider a heavy sub-combination between sub-components $x_1, \ldots, x_{i-1}$ and x_i (recall that since the subcombination is heavy, we have that $s(x_i) = p$). If each of $x_1, \ldots, x_{i-1}$ contains fewer than p edges, then we deduct p credits from each representative edge in each of $x_1, \ldots, x_{i-1}$. (Note that this is actually all of the edges in $x_1, \ldots, x_{i-1}$.) If at least one of $x_1, \ldots, x_{i-1}$ contains p or more edges, then we deduct p credits from each representative edge of x_i. In both cases, we deduct at least $\min\left(\sum_{j=1}^{\lfloor i/2 \rfloor} s(x_{2j}), p\right) \cdot s(x_i)$ credits in total, corresponding with the work done during the sub-combination.

The deductions of credits are designed so that two important properties hold: (1) whenever an edge e has credits deducted during a multi-spawn combination, the edge e will no longer be a representative edge in the new component C constructed by the multi-spawn combination; and (2) within a multi-spawn combination each edge e will have credits deducted from it at most once. Combined, these properties ensure that each edge has credits deducted at most once during the full construction of G. This, in turn, ensures that the total number of credits deducted by the algorithm is at most $|E| \cdot p$, and that the contribution of heavy sub-combinations to Equation (7.11) is also at most $|E| \cdot p$.

□

7.10 Appendix: An offline approximate-threshold algorithm

In this section we present our algorithm for the approximate threshold problem in the simpler offline setting, in which rather than supporting multi-spawn combinations, our recursive algorithm needs only support series and parallel combinations.

Our algorithm for the approximate threshold problem will compute the high-water mark over a special class of antichains that satisfy a certain property that we call robustness. (This is similar to the notion of stripped robustness from Section 7.5, except without any requirements about non-critical components; there are also several other minor differences designed to yield the simplest possible final algorithm.) The return value of the algorithm will then be determined by whether the computed value h is greater than $M/2$. In this section, we define what it means for an antichain to be robust and prove the correctness of our algorithm. Then, in Section 7.10, we describe a recursive algorithm for computing the quantity h needed by the algorithm in linear time $O(|E|)$.

When considering an antichain $A = (x_1, \ldots, x_q)$, we partition the predecessors of the antichain, $\{e \mid e < x_i \text{ for some } i\}$, into two categories. The ***core predecessors*** $\mathcal{C}(A)$ of the antichain A is the set of edges that are predecessors to more than one member of the antichain,

$$\mathcal{CP}(A) = \{e \mid e < x_i, e < x_j \text{ for some } i \neq j\}.$$

If an edge e is a predecessor of A but not a core predecessor, then e is a ***local predecessor*** of some x_i. We denote the set of local predecessors of x_i by

$$\mathcal{LP}_A(x_i) = \{e \mid e < x_i \text{ and } e \nless x_j \forall j \neq i\}.$$

We define the ***core companions*** $\mathcal{CC}(A)$ of the antichain A to be the set of edges e contained in a parallel component T_1 with positive edge sum and whose partnering parallel component T_2 contains multiple edges from the antichain A. For each x_i, we define the ***local companions*** $\mathcal{LC}_A(x_i)$ of x_i to be the set of edges e in a parallel component T_1 with positive edge sum and whose partnering parallel component T_2 contains the edge x_i but not any other edge $x_j \in A$.

The ***core water mark*** $C(A)$ is the sum of the edge totals over all edges in the core predecessors and companions of A,

$$C(A) = \sum_{e \in \mathcal{CP}(A) \cup \mathcal{CC}} t(e).$$

Similarly, the ***local water mark*** $L_A^\circ(x_i)$ of each edge $x_i \in A$ is given by

$$L_A^\circ(x_i) = m(x_i) + \sum_{e \in \mathcal{LP}_A(x_i) \cup \mathcal{LC}_A(x_i)} t(e).$$

The total water mark of the antichain can be rewritten as

$$W(A) = C(A) + \sum_{i=1}^{q} L_A^\circ(x_i).$$

Our algorithm for the approximate threshold problem will compute the infinite-processor high-water mark, except restricted only to antichains A whose local water marks all exceed $\frac{M}{2p}$. We call an antichain $A = (x_1, \ldots, x_q)$ ***robust*** if $L_A^\circ(x_i) > \frac{M}{2p}$ for each edge x_i. The ***p-processor robust memory high-water mark*** $H_p^\circ(G)$ is defined to be

$$H_p^\circ(G) = \max_{A \in \mathcal{R},\ |A| \leq p} W(x_1, \ldots, x_q),$$

where $\mathcal{R}$ is the set of robust antichains in E.

The first step in our algorithm will be to compute the infinite-processor robust memory high-water mark $H_\infty^\circ(G)$. Then, if $H_\infty^\circ(G) \leq M/2$, our algorithm will return 0, and if $H_\infty^\circ(G) > M/2$, our algorithm will return 1.

Computing $H_\infty^\circ(G)$ can be done in linear time $O(|E|)$ using a recursive algorithm described in Section 7.10. The computation is made significantly easier, in particular, by the fact that it is permitted to consider the infinite-processor case rather than restricting to p processors or fewer.

On the other hand, the fact that $H_\infty^\circ(G)$ should tell us anything useful about $H_p(G)$ is non-obvious. In the rest of this section, we will prove the following theorem, which implies the correctness of the algorithm:

Theorem 40

- *If $H_\infty^\circ(G) \leq M/2$, then $H_p(G) \leq M$.*

- *If $H^\circ_\infty(G) > M/2$, then $H_p(G) > M/2$.*

To prove Theorem 40, we begin by comparing $H^\circ_p(G)$ and $H_p(G)$:

Lemma 41
$$H^\circ_p(G) \geq H_p(G) - \frac{M}{2}.$$

Proof. Consider an antichain $A = (x_1, \ldots, x_q)$, with $q \leq p$, that is not robust. One might try to construct a robust antichain B by removing each $x_i \in A$ that satisfies $L^\circ_A(x_i) \leq \frac{M}{2p}$. The removal of these x_i's, however, would change the sets of local predecessors and companions for the remaining x_j's, making it so that the new antichain B may still not be robust.

One can instead obtain a robust antichain through a more iterative approach. Begin with the antichain $A_1 = A$ that is not robust. Since A_1 is not robust, some $x_i \in A$ satisfies $L^\circ_A(x_i) \leq \frac{M}{2p}$. Define A_2 to be the same antichain with x_i removed. If the antichain A_2 is also not robust, then pick some edge $x_j \in A_2$ such that $L^\circ_{A_2}(x_j) \leq \frac{M}{2p}$, and define A_3 to be A_2 with x_j removed. Continue like this until we reach some A_r that is robust. (Note that one legal option for A_r is the empty antichain, which is considered to be robust.)

For any two consecutive antichains A_i and A_{i+1} in the sequence, that differ by the removal of an edge x_j, the water marks satisfy

$$W(A_{i+1}) \geq W(A_i) - L^\circ_{A_i}(x_j). \tag{7.12}$$

(Note that the reason that Equation (7.12) is not true with equality is simply that the removal of x_j may allow for the addition of a new companion component to the antichain A_{i+1}, thereby making $W(A_{i+1})$ greater than $W(A_i) - L^\circ_{A_i}(x_j)$.)

Since we only remove edges x_j satisfying $L^\circ_{A_i}(x_j) \leq \frac{M}{2p}$, it follows that

$$W(A_{i+1}) \geq W(A_i) - \frac{M}{2p}.$$

Moreover, in the processes of constructing the robust antichain A_r, we can remove a total of at most q edges from the original antichain $A = (x_1, \ldots, x_q)$. Thus

$$W(A_r) \geq W(A) - q \cdot \frac{M}{2p} \geq W(A) - \frac{M}{2}.$$

This, in turn, implies that $H^\circ_p(G) \geq H_p(G) - \frac{M}{2}$, as desired. □

The following corollary proves the first part of Theorem 40

Corollary 42 *If $H^\circ_\infty(G) \leq M/2$, then $H_p(G) \leq M$.*

Proof. If $H^\circ_\infty(G) \leq M/2$, then $H^\circ_p(G) \leq M/2$, and thus by Lemma 41, $H_p(G) \leq M$. □

The second half of Theorem 40 is given by Lemma 43:

Lemma 43 *If $H^\circ_\infty(G) > M/2$, then $H_p(G) > M/2$.*

Proof. Since $H^\circ_\infty(G) > M/2$, one of the following must be true:

- **There is a robust antichain $A = (x_1, \ldots, x_q)$ with $q \leq p$ such that $W(A) > M/2$:** In this case, we trivially get that $H_p(G) > M/2$.

- **There is a robust antichain $A = (x_1, \ldots, x_q)$ with $q > p$ such that $W(A) > M/2$:** This case is somewhat more subtle, since the large number of edges in the antichain A could cause $W(A)$ to be much larger than $H_p(G)$. We will use the robustness of A in order to prove that the potentially much smaller antichain $B = (x_1, \ldots, x_p)$ still has a large water mark $W(B) > \frac{M}{2}$.

  Let $\mathcal{T}$ denote the set of edges $e \in E$ such that either $e \leq x_i$ for some $i \in [p]$, or e is contained in a companion parallel component of B. The quantity $W(B)$ can be written as

$$\begin{aligned} W(B) &= \sum_{e \in \mathcal{T}} t(e) \\ &= \sum_{i=1}^{p} L_A^\circ(x_i) + \sum_{e \in \mathcal{T} \cap (\mathcal{CP}(A) \cup \mathcal{CC}(A))} t(e). \end{aligned}$$

  By the robustness of A, each local water mark $L_A^\circ(x_i)$ is greater than $\frac{M}{2p}$. Thus

$$W(B) > \frac{M}{2} + \sum_{e \in \mathcal{T} \cap (\mathcal{CP}(A) \cup \mathcal{CC}(A))} t(e).$$

  Recall the downset-non-negativity property, which requires that every downset $S \subseteq E$ (meaning that the predecessors of the edges in S are all in S) satisfy $\sum_{e \in S} t(e) \geq 0$. To see that the $\mathcal{T}$ is a downset, observe that it consists of two parts, the set $\mathcal{T}_1$ of predecessors of B, and the set $\mathcal{T}_2$ of edges contained in companion parallel components to B; since the set $\mathcal{T}_1$ is a downset, and because the predecessors of edges in $\mathcal{T}_2$ are all either in $\mathcal{T}_2$ or in $\mathcal{T}_1$, the full set $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$ is a downset. Similarly we claim that $\mathcal{CP}(A) \cup \mathcal{CC}(A)$ is a downset; in particular, $\mathcal{CP}(A)$ is a downset by its definition, and the predecessors of edges in $\mathcal{CC}(A)$ are all either contained in $\mathcal{CC}(A)$ or in $\mathcal{CP}(A)$. Since we have shown that $\mathcal{T}$ and $\mathcal{CP}(A) \cup \mathcal{CC}(A)$ are downsets, and because the intersection of two downsets is necessarily also a downset, it follows that $\mathcal{T} \cap (\mathcal{CP}(A) \cup \mathcal{CC}(A))$ is a downset.

  Applying the downset-non-negativity property, we get that

$$\sum_{e \in \mathcal{T} \cap (\mathcal{CP}(A) \cup \mathcal{CC}(A))} t(e) \geq 0,$$

  implying that $W(B) > \frac{M}{2}$, and completing the proof.

□

Computing $H_\infty^\circ(G)$ in linear time

In this section, we present a recursive algorithm for computing the infinite-processor robust high-water mark $H_\infty^\circ(G)$ in linear time $O(|E|)$. We assume that we are given the series-parallel DAG G, along with the labels $t(e)$ and $m(e)$ for each $e \in G$.

Suppose we recursively build G from series and parallel combinations. Whenever we create a new component C (by combining two old ones) we will maintain the following information on the component:

- The total allocation and freeing work done in C,
$$\text{MemTotal} = \sum_{e \in C} t(e).$$
- The memory high-water mark with one processor,
$$\text{MaxSingle} = H_1(C).$$
- The infinite-processor memory high-water mark restricted only to robust antichains containing more than one edge:
$$\text{MultiRobust} = \max_{A \in \mathcal{R},\ |A|>1} W(A),$$
where $\mathcal{R}$ is the set of robust antichains in G. If C contains no multi-edge robust antichains, then MultiRobust = null, and is treated as $-\infty$.

As a base case, for a component C consisting of a single edge e, we initialize the variables as follows: MemTotal $= t(e)$, MaxSingle $= m(e)$, and MultiRobust = null.

When we combine two components C_1 and C_2 in series to build a new component C, the three variables can be updated as follows:

- We update $C.$ MemTotal as
$$C_1.\text{MemTotal} + C_2.\text{MemTotal}.$$
In particular, $\sum_{e \in C} t(e) = \sum_{e \in C_1} t(e) + \sum_{e \in C_2} t(e)$.
- We update $C.$ MaxSingle as
$$\max(C_1.\text{MaxSingle}, C_1.\text{MemTotal} + C_2.\text{MaxSingle}).$$
In particular, every single-edge antichain in C_1 has the same water mark in C as it did in C_1, and every single-edge antichain in C_2 has cost in C an additional $C_1.$ MemTotal greater than it did in C_2.
- We update $C.$ MultiRobust as
$$\max(C_1.\text{MultiRobust}, C_1.\text{MemTotal} + C_2.\text{MultiRobust}).$$
In particular, the set of multi-edge robust antichains in the new component C is the union of the set of multi-edge antichains in C_1 with the set of multi-edge antichains in C_2. Whereas each of the multi-edge antichains in C_1 have the same water mark in C as they did in C_1, the multi-edge antichains in C_2 each have their water marks increased by $C_1.$ MemTotal.

When we combine two components C_1 and C_2 in parallel to build a new component C, the three variables can be updated as follows:

- We update $C.\operatorname{MemTotal}$ as
$$C_1.\operatorname{MemTotal} + C_2.\operatorname{MemTotal}.$$
In particular, just as before, $\sum_{e\in C} t(e) = \sum_{e \in C_1} t(e) + \sum_{e\in C_2} t(e)$.

- We update $C.\operatorname{MaxSingle}$ as
$$\max(C_1.\operatorname{MaxSingle} + \max(0, C_2.\operatorname{MemTotal}),\\ C_2.\operatorname{MaxSingle} + \max(0, C_1.\operatorname{MemTotal})).$$
In particular, the set of single-edge antichains in C is the union of the set of single-edge antichains in C_1 with the set of single-edge antichains in C_2. Since the water marks of the antichains are the same in C_1 and C_2 as they are in C, except with the addition of $\max(0, C_2.\operatorname{MemTotal})$ and $\max(0, C_1.\operatorname{MemTotal})$ respectively (due to the possibility of C_2 and C_1 being suspended companion parallel components), $C.\operatorname{MaxSingle}$ can be updated by taking a simple maximum of the two options.

- The update of $C.\operatorname{MultiRobust}$ is slightly more subtle. Define $C_1.\overline{\text{MaxSingle}}$ and $C_2.\overline{\text{MaxSingle}}$ to be the highest water marks achieved by robust single-edge antichains in C_1 and C_2, respectively. That is,
$$C_1.\overline{\text{MaxSingle}} = \begin{cases} C_1.\operatorname{MaxSingle} \text{ if } C_1.\operatorname{MaxSingle} > \frac{M}{2p} \\ \text{null otherwise,} \end{cases}$$
and
$$C_2.\overline{\text{MaxSingle}} = \begin{cases} C_2.\operatorname{MaxSingle} \text{ if } C_2.\operatorname{MaxSingle} > \frac{M}{2p} \\ \text{null otherwise.} \end{cases}$$
Define $\mathcal{R}(C)$, $\mathcal{R}(C_1)$, and $\mathcal{R}(C_2)$ to be the sets of robust antichains in C, C_1, and C_2, respectively. Then, because C is obtained by combining C_1 and C_2 in parallel,
$$\mathcal{R}(C) = \{x \cup y \mid x \in \mathcal{R}(C_1), y \in \mathcal{R}(C_2)\}.$$
When computing $C.\operatorname{MultiRobust}$, we are interested exclusively in the antichains A satisfying $|A| > 1$. If $x = A \cap C_1$ and $y = A \cap C_2$, then the requirement that $|A| > 1$ translates into the requirement that (at least) one of the following three requirements holds:

  1. $|x| = |y| = 1$: The maximum water mark for robust antichains a such that $|x| = |y| = 1$ is given by
  $$C_1.\overline{\text{MaxSingle}} + C_2.\overline{\text{MaxSingle}}.$$
  2. $|y| > 1$: The maximum water mark for robust antichains a such that $|y| > 1$ is given by
  $$\max(C_1.\overline{\text{MaxSingle}}, C_1.\operatorname{MultiRobust}, C_1.\operatorname{MemTotal}, 0)\\ + C_2.\operatorname{MultiRobust},$$
  where entries in the maximum correspond with the cases where $|x| > 1$; $|x| = 1$;

$|x|= 0$ and C_1 is included as a suspended companion component; and $|x|= 0$ and C_1 is not included as a suspended companion component.

3. $|x|> 1$: The maximum water mark for robust antichains a such that $|x|> 1$ is given by

$$\max(C_2.\overline{\text{MaxSingle}}, C_2.\text{MultiRobust}, C_2.\text{MemTotal}, 0) \\ + C_1.\text{MultiRobust},$$

where entries in the maximum correspond with the cases where $|y|> 1$; $|y|= 1$; $|y|= 0$ and C_2 is included as a suspended companion component; and $|y|= 0$ and C_2 is not included as a suspended companion component.

Combining the cases, we can update $C.\text{MultiRobust}$ as

$$\max\Big(C_1.\overline{\text{MaxSingle}} + C_2.\overline{\text{MaxSingle}}, \\ \max(C_1.\overline{\text{MaxSingle}}, C_1.\text{MultiRobust}, \\ C_1.\text{MemTotal}, 0) + C_2.\text{MultiRobust}, \\ \max(C_2.\overline{\text{MaxSingle}}, C_2.\text{MultiRobust}, \\ C_2.\text{MemTotal}, 0) + C_1.\text{MultiRobust} \Big).$$

Using the recursive construction described above, we can compute the variables MemTotal, MaxSingle, and MultiRobust for our graph G in linear time $O(|E|)$. In order to then compute $H^{\circ}_{\infty}(G)$, the infinite-processor high-water mark considering only robust antichains, we simply compute

$$H^{\circ}_{\infty}(G) = \max\left(\left\{ \begin{array}{l} \text{MaxSingle if MaxSingle} > \frac{M}{2p} \\ \text{null otherwise} \end{array} \right\}, \text{MultiRobust}, 0\right).$$

7.11 Appendix: Recursing on multi-spawn components

In this section, we complete the recursion for $H^{\bullet}_{\infty}(G)$ discussed in Section 7.5.2 by handling the case of multi-spawn combinations.

Consider a multi-spawn combination C as in Figure 7-2 with components $C_1 = a_0, C_2 = b_1, C_3 = a_1, \ldots, C_{2k+1} = a_k$.

Throughout the section, we will use the notation $m(b_i)$ and $t(b_i)$ introduced in Lemma 37. When b_i is the first case of the lemma, we say that b_i ***is a natural companion*** and that b_i's ***natural contribution*** is $t(b_i)$; when b_i is in the second case, we say that b_i ***is naturally dormant*** and that b_i's ***natural contribution*** is 0; when b_i is in the third case, we say that b_i ***is naturally active*** and that b_i's ***natural contribution*** is $m(b_i)$.

Recall that the execution of the parallel program on one thread computes the recursive values for each C_i with i iterating through the range $i = 1, \ldots, 2k+1$. We wish to use these in order to compute the recursive values for C.

To do this, we maintain a collection of intermediate values during the execution of the components $C_1, \ldots, C_{2k+1}$. Before introducing these intermediate values, we define a few terms.

We call a stripped robust antichain A in C a ***candidate*** antichain if for each b_i in C such that A contains an edge in one of $a_i, b_{i+1}, a_{i+1}, \ldots$, the three properties stated in

Lemma 37 hold. (In particular, b_i's local contribution $L^{\bullet}_A(b_i)$ should be precisely b_i's natural contribution.) By Lemma 37, when computing computing $C.\text{MultiRobust}$, it suffices to consider only multi-edge candidate antichains.

In order to describe the intermediate values that we maintain during the execution of the components, we will also need the notion of a suspendend-end and ignored-end water mark. (These are the same definitions as used in Section 7.9.) If an antichain A in C contains only edges in $b_1, \ldots, b_k$, and b_t is the largest t such that b_t contains a edge in A, then we say A has a ***suspended end*** if the components $a_{t+1}, b_{t+2}, \ldots, a_k$ form a companion component of A (which occurs if the sum of their edge costs is net positive). The ***suspended-end water mark*** of A is $W(A)$ if A has a suspended end, and is $W(A) + \sum_{e \in \bigcup a_{t+1}, b_{t+2}, \ldots, a_k} t(e)$ if A does not have a suspended end (i.e., it is the water mark A would have if it had a suspended end). Similarly, the ***ignored-end water mark*** of A is $W(A)$ if A does not have a suspended end, and is $W(A) - \sum_{e \in \bigcup a_{t+1}, b_{t+2}, \ldots, a_k} t(e)$ if A does have a suspended end. These definitions will prove useful when defining the intermediate values maintained by our algorithm. Additionally, for any antichain A that contains an edge in some a_t, we define the ***ignored-end water mark*** of A to be the water mark of A; thus the ignored-end water mark is defined for all antichains in A of C.

After having executed each of $C_1, \ldots, C_l$, let l_1 be the index of the largest-indexed a_i executed and l_2 be the index of the largest-indexed b_i executed. We maintain the following intermediate values:

- $\text{MultiRobustSuspendEnd}_l$: This is the maximum suspended-end cost of any multi-edge candidate antichain A in $C_1 \cup \cdots \cup C_l$ containing only edges in $b_1, \ldots, b_{l_2}$. If no such A exists, this is null. Note that here $C_1 \cup \cdots \cup C_l$ is treated as a multi-spawn component and the costs of the antichains are considered just within the graph $C_1 \cup \cdots \cup C_l$, rather than the full graph C (which matters because we are considering the suspended-end cost of the antichain).

- $\text{MultiRobustIgnoreEnd}_l$: This is the maximum ignored-end cost of any multi-edge candidate antichain A in $C_1 \cup \cdots \cup C_l$. If no such A exists, this is null.

- $\text{SingleSuspendEnd}_l$: This is the maximum suspended-end cost of any single-edge antichain A in $C_1 \cup \cdots \cup C_l$ such that A contains only edges in $b_1, \ldots, b_{l_2}$. (Again, we consider the suspended-end cost just within the graph $C_1 \cup \cdots \cup \cdots \cup C_l$.)

- SingleIgnoreEnd_l: This is the maximum ignored-end cost of any single-edge antichain A in $C_1 \cup \cdots \cup C_l$.

- $\text{RobustUnfinished}_l$: Let t be the largest $t \le l_2$ such that b_t is naturally active. Then $\text{RobustUnfinished}_l$ is the sum of the natural contributions of $b_1, \ldots, b_t$, along with $t(a_0), t(a_1) + \cdots + t(a_{t-1})$. If no such t exists, then $\text{RobustUnfinished}_l$ is null.

  One should think of this as the contribution of $b_1, \ldots, b_t$ and $a_0, \ldots, a_{t-1}$ to any candidate antichain in C that contains at least one edge in $b_{l_2+1}, \ldots, b_k$ or $a_{l_1+1}, \ldots, a_k$. (We separate this from the contribution of the edges $b_{t+1}, \ldots, b_{l_2}$ and $a_t, \ldots, a_{l_1}$ which are considered by the next quantity.)

- $\text{RobustUnfinishedTail}_l$: Let t be the largest $t \le l$ such that b_t is naturally active, or 0 if no such t exists. Then $\text{RobustUnfinishedTail}_l$ is the sum of the natural contributions of $b_{t+1}, \ldots, b_{l_2}$, along with $t(a_t) + t(a_{t+1}) + \cdots + t(a_{l_1})$.

One should think of this as the contribution of $b_{t+1}, \ldots, b_{l_2}$ and $a_t, \ldots, a_{l_1}$ to any candidate antichain in C that contains at least one edge in $b_{l_2+1}, \ldots, b_k$ or $a_{l_1+1}, \ldots, a_k$. The quantity $\text{RobustUnfinishedTail}_l$ is handled separately from $\text{RobustUnfinished}_l$ because if the candidate antichain contains only a single edge in $b_{l_2+1}, \ldots, b_k$ or $a_{l_1+1}, \ldots, a_k$, then $\text{RobustUnfinishedTail}_l$ can affect the local contribution of that edge.

- RunningMemTotal_l: This is $\sum_{i=0}^{l_1} t(a_i) + \sum_{i=1}^{l_2} t(b_i)$, the total sum of the edge totals over all edges in the components $a_0, \ldots, a_{l_1}, b_1, \ldots, b_{l_2}$.

- EmptyTail_l: This is $\sum_{i=0}^{l_1} t(a_i) + \sum_{i=1}^{l_2} \max(0, t(b_i))$. One should think of this as the contribution of $a_0, \ldots, a_{l_1}, b_1, \ldots, b_{l_2}$ to any single-edge antichain in C whose edge lies in one of $a_{l_1+1}, a_{l_1+2}, \ldots$ or $b_{l_2+1}, b_{l_2+2}, \ldots$.

Given the above variables for $l = 2k+1$, one can compute

$$C.\text{MemTotal} = \text{RunningMemTotal}_{2k+1},$$

$$C.\text{MultiRobust} = \max(\text{MultiRobustSuspendEnd}_{2k+1}, \text{MultiRobustIgnoreEnd}_{2k+1}),$$

and

$$C.\text{MaxSingle} = \max(\text{SingleSuspendEnd}_{2k+1}, \text{SingleIgnoreEnd}_{2k+1}).$$

Prior to beginning, we have $l = 0$, and have that $\text{MultiRobustSuspendEnd}_0 = \text{null}$, $\text{MultiRobustIgnoreEnd}_0 = \text{null}$, $\text{SingleSuspendEnd}_0 = \text{null}$, $\text{SingleIgnoreEnd}_0 = \text{null}$, $\text{RobustUnfinished}_0 = \text{null}$, $\text{RobustUnfinishedTail}_0 = 0$, $\text{RunningMemTotal}_0 = 0$, and $\text{EmptyTail}_0 = 0$.

To complete the algorithm, we present the protocol for advancing l by one, and updating each of the intermediate values.

Suppose for some odd $l > 0$ we are given the values of the above quantities for $l-1$, and given the recursive values for $a_{(l+1)/2}$. We obtain the new values for l as follows:

- **Step 1: Simple Updates.** We compute $\text{MultiRobustSuspendEnd}_l$ as,

  $$\text{MultiRobustSuspendEnd}_{l-1} + a_{(l+1)/2}.\text{MemTotal},$$

  and $\text{SingleSuspendEnd}_l$ as,

  $$\text{SingleSuspendEnd}_{l-1} + a_{(l+1)/2}.\text{MemTotal}.$$

  We compute SingleIgnoreEnd_l as

  $$\max(\text{SingleIgnoreEnd}_{l-1}, a_{(l+1)/2}.\text{MaxSingle} + \text{EmptyTail}_{l-1}),$$

  where the second entry in the maximum is the largest water mark of any single-edge antichain in C with an edge in $a_{(l+1)/2}$.

We set $\text{RobustUnfinished}_l = \text{RobustUnfinished}_{l-1}$. Finally we increase the following $\text{RobustUnfinishedTail}_l$, RunningMemTotal_l, and EmptyTail_l by $a_{(l+1)/2}.\text{MemTotal}$ over their values for $l-1$ (where the outcome is null if they were previously null).

- **Step 2: Computing** $\text{MultiRobustIgnoreEnd}_l$**.** We update $\text{MultiRobustIgnoreEnd}_l$ with Algorithm 3. The only antichains A that $\text{MultiRobustIgnoreEnd}_l$ needs to consider but that $\text{MultiRobustIgnoreEnd}_{l-1}$ did not are the candidate stripped robust antichains A containing at least one edge in $a_{(l+1)/2}$.

  The first if-statement checks whether any multi-edge candidate antichains exist in which $a_{(l+1)/2}$ contributes only a single edge; this requires that $\text{RobustUnfinishedTail}_{l-1} + a_{(l+1)/2}.\text{MaxSingle} > \frac{M}{2p}$ in order for the local contribution of the edge in $a_{(l+1)/2}$ to exceed $\frac{M}{2p}$; and that $\text{RobustUnfinished} \neq \text{null}$ that way the resulting antichain contains multiple edges.

  The second if-statement considers candidate antichains in which $a_{(l+1)/2}$ contributes multiple edges. If $\text{RobustUnfinished}_{l-1} \neq \text{null}$, then the maximum water mark in C obtainable by such an antichain is $\text{RobustUnfinished}_{l-1} + \text{RobustUnfinishedTail}_{l-1}$ $+a_{(l+1)/2}.\text{MultiRobust}$. If $\text{RobustUnfinished}_{l-1} = \text{null}$, then the maximum water mark in C obtainable is $\text{RobustUnfinishedTail}_{l-1} + a_{(l+1)/2}.\text{MultiRobust}$.

Suppose for some even $l > 0$ we are given the values of the intermediate values for $l-1$, and given the recursive values for $b_{l/2}$. We obtain the new values for l as follows:

- **Step 1: Simple Updates:** We compute $\text{SingleSuspendEnd}_l$ as

$$\max(\text{SingleSuspendEnd}_{l-1} + b_{l/2}.\text{MemTotal}, \\ b_{l/2}.\text{MaxSingle} + \text{EmptyTail}_{l-1}),$$

  and SingleIgnoreEnd_l as,

$$\max(\text{SingleIgnoreEnd}_{l-1}, b_{l/2}.\text{MaxSingle} + \text{EmptyTail}_{l-1}).$$

  We compute RunningMemTotal_l as,

$$\text{RunningMemTotal}_{l-1} + b_{l/2}.\text{MemTotal}\,.$$

  Finally, we compute EmptyTail_l as,

$$\text{EmptyTail}_l = \text{EmptyTail}_{l-1} + \max(0, b_{l/2}.\text{MemTotal}).$$

- **Step 2: Computing** $\text{MultiRobustSuspendEnd}_l$ **and** $\text{MultiRobustIgnoreEnd}_l$**.** We update $\text{MultiRobustSuspendEnd}_l$ and $\text{MultiRobustIgnoreEnd}_l$ with Algorithm 4. We begin by computing X, the largest ignored-end cost of any candidate stripped robust antichain in C that (1) contains multiple edges; (2) contains at least one edge in $b_{l/2}$; and (3) contains no edges in $a_{l/2}, b_{l/2+1}, \ldots, a_k$. The first if-statement considers the case where the antichain has one edge in $b_{l/2}$; and the second considers the case where there are multiple such edges.

  After computing X, we update $\text{MultiRobustSuspendEnd}_l$ and $\text{MultiRobustIgnoreEnd}_l$ based on X's value.

- **Step 3: Computing** $\text{RobustUnfinished}_l$ **and** $\text{RobustUnfinishedTail}_l$**.** We compute $\text{RobustUnfinished}_l$ and $\text{RobustUnfinishedTail}_l$ with Algorithm 5. We define m and t to be $m(b_{l/2})$ and $t(b_{l/2})$, as defined in Lemma 37. We then update $\text{RobustUnfinished}_l$ and $\text{RobustUnfinishedTail}_l$ appropriately based on the three cases in the lemma. (In the final case, we take the maximum of 0 and $\text{RobustUnfinished}_{l-1}$ because if the latter is null, we wish to treat it as zero.)

This completes the recursion described in Section 7.5.2, allowing one to compute $H^{\bullet}_{\infty}(G)$ in an online manner (i.e., while executing the parallel program on a single thread) with constant time and space overhead.

Algorithm 3 Updating MultiRobustIgnoreEnd for $a_{(l+1)/2}$

```
$\text{MultiRobustIgnoreEnd}_l = \text{MultiRobustIgnoreEnd}_{l-1}$;
if $\text{RobustUnfinishedTail}_{l-1} + a_{(l+1)/2}.\text{MaxSingle} > \frac{M}{2p}$ and $\text{RobustUnfinished}_{l-1} \neq \text{null}$
    $\text{MultiRobustIgnoreEnd}_l = \max(\text{self}, \text{RobustUnfinished}_{l-1} + \text{RobustUnfinishedTail}_{l-1} + a_{(l+1)/2}.\text{MaxSingle})$;
if $a_{(l+1)/2}.\text{MultiRobust} \neq \text{null}$
    if $\text{RobustUnfinished}_{l-1} \neq \text{null}$
        $\text{MultiRobustIgnoreEnd}_l = \max(\text{self}, \text{RobustUnfinished}_{l-1} + \text{RobustUnfinishedTail}_{l-1} + a_{(l+1)/2}.\text{MultiRobust})$;
    if $\text{RobustUnfinished}_{l-1} = \text{null}$
        $\text{MultiRobustIgnoreEnd}_l = \max(\text{self}, \text{RobustUnfinishedTail}_{l-1} + a_{(l+1)/2}.\text{MultiRobust})$;
```

Algorithm 4 Updating MultiRobustSuspendEnd and MultiRobustIgnoreEnd for $b_{l/2}$

```
$X = \text{null}$;
if $b_{l/2}.\text{MaxSingle} + \text{RobustUnfinishedTail}_{l-1} > \frac{M}{2p}$ and $\text{RobustUnfinished}_{l-1} \neq \text{null}$
    $X = b_{l/2}.\text{MaxSingle} + \text{RobustUnfinishedTail}_{l-1} + \text{RobustUnfinished}_{l-1}$;
if $b_{l/2}.\text{MultiRobust} \neq \text{null}$
    if $\text{RobustUnfinished}_{l-1} \neq \text{null}$
        $X = \max(X, \text{RobustUnfinished}_{l-1} + \text{RobustUnfinishedTail}_{l-1} + b_{l/2}.\text{MultiRobust})$
    if $\text{RobustUnfinished}_{l-1} = \text{null}$
        $X = \max(X, \text{RobustUnfinishedTail}_{l-1} + b_{l/2}.\text{MultiRobust})$
$\text{MultiRobustSuspendEnd}_l = \max(X, \text{MultiRobustSuspendEnd}_{l-1})$;
$\text{MultiRobustIgnoreEnd}_l = \max(X, \text{MultiRobustIgnoreEnd}_{l-1})$;
```

Algorithm 5 Updating RobustUnfinished and RobustUnfinishedTail for $b_{l/2}$

```
$m = b_{l/2}.\text{MultiRobust}$;
if $b_{l/2}.\text{MaxSingle} > \frac{M}{2p}$
    $m = \max(m, b_{l/2}.\text{MaxSingle})$;
if $m = \text{null}$
    $m = 0$;
$t = b_{l/2}.\text{MemTotal}$;
if $t > 0$ and $m \leq t + \frac{M}{2p}$
    $\text{RobustUnfinished}_l = \text{RobustUnfinished}_{l-1}$;
    $\text{RobustUnfinishedTail}_l = \text{RobustUnfinishedTail}_{l-1} + t$;
if $t \leq 0$ and $m \leq \frac{M}{2p}$
    $\text{RobustUnfinished}_l = \text{RobustUnfinished}_{l-1}$;
    $\text{RobustUnfinishedTail}_l = \text{RobustUnfinishedTail}_{l-1}$;
if $m \geq \max(0, t) + \frac{M}{2p}$
    $\text{RobustUnfinished}_l = \max(0, \text{RobustUnfinished}_{l-1}) + \text{RobustUnfinishedTail}_{l-1} + m$;
    $\text{RobustUnfinishedTail}_l = 0$;
```

Chapter 8

Optimal Reissue Policies for Reducing Tail-Latency

This chapter presents work on the design and formulation of the SingleR policy family for reducing tail-latency in distributed request-response workflows by judiciously sending redundant requests. The work presented in this chapter was conducted in collaboration with Yuxiong He and Sameh Elnikety.

Abstract

Interactive services send redundant requests to multiple different replicas to meet stringent tail latency requirements. These additional (reissue) requests mitigate the impact of non-deterministic delays within the system and thus increase the probability of receiving an on-time response.

There are two existing approaches of using reissue requests to reduce tail latency. (1) Reissue requests immediately to one or more replicas, which multiplies the load and runs the risk of overloading the system. (2) Reissue requests if not completed after a fixed delay. The delay helps to bound the number of extra reissue requests, but it also reduces the chance for those requests to respond before a tail latency target.

We introduce a new family of reissue policies, **Single-Time / Random** (SingleR), that reissue requests after a delay d with probability q. SingleR employs randomness to bound the reissue rate, while allowing requests to be reissued early enough so they have sufficient time to respond, exploiting the benefits of both immediate and delayed reissue of prior work. We formally prove, within a simplified analytical model, that SingleR is optimal even when compared to more complex policies that reissue multiple times.

To use SingleR for interactive services, we provide efficient algorithms for calculating optimal reissue delay and probability from response time logs through a data-driven approach. We apply iterative adaptation for systems with load-dependent queuing delays. The key advantage of this data-driven approach is its wide applicability and effectiveness to systems with various design choices and workload properties.

We evaluated SingleR policies thoroughly. We use simulation to illustrate its internals and demonstrate its robustness to a wide range of workloads. We conduct system experiments on the Redis key-value store and Lucene search server. The results show that for utilizations ranging from 40-60%, SingleR reduces the 99th-percentile latency of Redis by 30-70% by reissuing only 2% of requests, and the 99th-percentile latency of Lucene by 15-25% by reissuing 1% only.

8.1 Introduction

Interactive online services, such as web search, financial trading, and games require consistently low response times to attract and retain users [144, 308]. The service providers therefore define strict targets for *tail latencies* — 95th percentile, 99th percentile or higher response times [85, 87, 148, 349] to deliver consistently fast responses to user requests. For many distributed and layered services, a request could span several servers and the responses are aggregated, in which case the slower servers typically dominate the response time [194]. As a result, tail latencies are more suitable performance metrics than averages in latency-sensitive applications that employ concurrency.

Variability in a service's response-time can lead to tail-latencies that are several orders of magnitude larger than the average or median. Rare work-intensive requests can have a disproportionate impact on tail-latency by causing other requests to be delayed. Other, often nondeterministic, factors also play a significant role: random load-balancing can lead to short-term skew between machines; background tasks on servers can lead to temporary shortages in CPU cycles, memory, and disk bandwidth; network congestion can increase latency and reduce throughput of communication channels. Reducing tail latency, influenced by all of these contributing factors, is challenging.

The judicious use of redundant computation is often a highly effective technique for reducing tail-latency in interactive services. The basic idea is to exploit inter-machine parallelism by sending multiple copies of a request to replicated servers in order to boost the probability of receiving at least one timely response. This technique is widely used by interactive services, yet despite its prevalence there has been little guidance on optimizing its usage.

We develop a methodology for designing reissue policies that is composed of 3 steps. First, we define several families of reissue policies of varied complexity. These reissue policies are parametrized by variables such as: a) whether to reissue a request, b) when to reissue a request, and c) how many times to reissue a request. We choose an optimal family of policies among the candidates guided by a theoretical analysis under a simplified model where the system's response-time distributions are static. Second, we provide an algorithm to find the optimal values for the policy's parameters using response-time logs, solving the constrained optimization problem efficiently. Third, we provide iterative algorithms for refining a policy's parameters in response to changes in system load, and for adjusting the total fraction of requests that are reissued to minimize tail-latency.

Related work and challenges.

This technique of reissuing latency-sensitive requests is not new. It has been employed by a wide variety of systems such as key-value stores [327, 67, 204, 347], distributed request-response workflows [176], DNS lookup [339, 7], TCP flows [348, 111], and web-search [85]. Existing systems that reissue requests to reduce tail-latency predominantly employ one of two strategies.

For systems that run at low utilization, the common approach is to perform *immediate reissue* of requests — i.e. dispatch multiple copies of all requests. The effectiveness of immediate reissue has been investigated in previous studies [327, 339, 348, 111]. The primary advantage of the immediate reissue approach is that all copies of a request have an equal chance to respond before a tail-latency deadline since they are dispatched at the same time. This advantage is a motivation within RepFlow [348] for employing immediate reissue for the replication of short TCP flows (under 100KB). The disadvantage of immediate reissue, however, is that its impact on overall load renders it ineffective for systems with moderate and high utilization. A recent study in [339] on memcached, for example, shows that immediate reissue can degrade performance at utilizations as low as 10%.

For systems that run at higher utilization, an alternative approach is to perform *delayed reissue* of requests [85, 83, 347, 176] — i.e. dispatch a second copy of a request after a delay d, which we refer to as **Single-Time / Deterministic** policy or SINGLED. The SINGLED policy family corresponds to the scheme proposed in "The Tail at Scale" by Dean and Barroso [85], where, for example, the delay d could be decided using 95th-percentile latency of the workload. The advantage of delayed reissue is that we save the cost of reissuing the requests that would respond fast anyway. However, if the delay d is picked to be too large, then there may not be sufficient time for a reissue request to respond before the latency target.

Along the line of analytical work, prior work only studied immediate reissues for average latency under very specific arrival/service time distributions. Joshi et al. [181, 182] study the impact of immediate reissuing on log-concave and log-convex service-time distributions. Gardner et al. [118] present an exact analysis of immediate reissue for poisson arrivals and exponential service-times. Lee et al. [214] consider minimizing average latency by reissuing requests with a known cancellation overhead. Shah et al. [310] analyze the effectiveness of immediate reissuing in the MDS queue model.

When it comes to developing effective reissue policies for reducing tail-latency on a wide range of workloads and systems, many questions remain largely unanswered. The problem is challenging for multiple reasons: (1) The impact of reissuing is complex: one must weigh the odds of reducing tail latency by sending a duplicate request against the increase in system utilization caused by adding load. (2) There is a large search space with many different choices of which requests to reissue and when. (3) The complex and different workload properties of various interactive services, such as service-time distributions, arrival patterns, request correlations, and system settings make it difficult to derive general strategies for reducing tail latency. (4) Analytical work using queueing theory is challenging even when making strong assumptions about response-time distributions (e.g. drawn from exponential family), and conclusions draw from such simple models are hard to generalize to more complex systems.

Methodology and key results.

The goal of our work is to find a reissue policy that minimizes a workload's kth percentile tail latency by issuing a fixed percentage (or budget) of redundant requests. We explore the space and devise reissue policies in a principled manner — directed by theoretical analysis to identify the key insights of effective reissue policies, and driven by empirical data from actual systems for wide applicability.

We introduce a new family of reissue policies, **Single-Time / Random** (SINGLER), that reissue requests after a delay d with probability q. The use of randomness in SINGLER provides an important degree of freedom that allows to bound the reissue budget while also ensuring that reissue requests have sufficient time to respond, exploiting the benefits of both immediate and delayed reissue of prior work.

Using a simplified analytical model, we formally prove that SINGLER is the optimal trade-off between the immediate and delayed reissue strategies. More precisely, we define the ***Multiple-Time / Random*** (MULTIPLER) policies which reissue requests multiple times with different delays and reissue probabilities. We prove that, surprisingly, the optimal policies in MultipleR and SingleR are equivalent. It is a powerful result, restraining the complexity of reissue policies to one time reissue only while guaranteeing the effectiveness of SingleR.

Next, we present how to apply SingleR for interactive services through a data-driven approach to efficiently find the appropriate parameters, reissue time and probability, given sampled response times of the workloads. Our approach takes into account correlations between primary and reissue request response times. It is computationally efficient, finding optimal values of the parameters in close to linear time, with respect to the data size.

Moreover, we show how to devise reissue policies for systems which are sensitive to added load by adaptively refining a reissue policy in response to feedback from the system. This method remains oblivious to many system design details, relies on iterative adaptation to discover a system's response-time distributions and its response to added load. This data-driven approach is performed in a principled manner: every refined policy is the solution to a well defined optimization problem based on updated response-time distributions, applicable to a wide range of workloads with varying properties.

Empirical evaluation

We illustrate the properties of SingleR using both simulation and system experiments. Through careful simulation, we illustrate two key points: 1) the use of randomization in SingleR is especially important for workloads with correlated service times and queueing delays, 2) the effectiveness of SingleR is robust to varied workload properties and design choices including: utilization, service-time distribution, target latency percentiles, service-time correlations, and load-balancing/request-prioritization strategies.

We also evaluate SingleR using two distributed systems based on Redis [358] and Lucene enterprise search [237]. We demonstrate that, on a wide range of utilizations from 20-60%, SingleR is able to reduce tail-latency significantly while reissuing only a small number of requests. Even at 40-60% utilization, which is high for interactive services, SingleR reduces the 99th-percentile latency of Redis by 30-70% while reissuing only 2% of requests, and the 99th-percentile latency of Lucene by 15-25% while reissuing just 1% of requests.

Summary of contributions

1. We introduce the SingleR reissue policy family that reissues requests after a delay d with probability q. It exploits randomness to permit the timely reissue of requests with bounded budget, achieving the benefits of both immediate and delayed reissue (Section 8.2).

2. We prove within a simplified analytical model that the optimal policies in MultipleR and SingleR are equivalent. Reissuing more than once does not offer additional benefit — SingleR is simple and effective. (Section 8.3).

3. We show how to apply SingleR for interactive services by providing efficient algorithms for obtaining reissue delay and probability parameters from response time logs. (Section 8.4).

4. We evaluate SingleR using both simulation and system experiments on Redis key value store and Lucene search server (Section 8.5 and Section 8.6).

Note that our methodology for developing reissue policies utilizes multiple performance models of increasing complexity. This is a strategic choice that allows us to make definitive design choices that are guided by theoretical insights. The proof that SingleR is optimal relative to SingleD and MultipleR operates in a simplified model in which policies reissue only a fixed fraction of requests, and where the service's response-time distributions are static and uncorrelated. This simplified model allows us to address questions about the general structure of reissue policies that are otherwise intractable. Our algorithms for finding the optimal SingleR policy for a specific interactive service operates in a less constrained model where response-times may be correlated. Our techniques for adaptively refining SingleR policies are in a more general model in which a system may have load-dependent queueing delays — i.e. reissue requests perturb the response-time distribution. The sequence of decisions made with respect to performance model are not arbitrary. As shown in the empirical analysis of SingleR on simulated workloads in Section 8.5 and in real-world systems in Section 8.6 these steps lead to effective reissue policies and the insights made in simpler models are readily recognizable in our empirical results.

8.2 Deterministic versus random reissue

In this section, we introduce the ***Single-Time / Random*** (SingleR) policies, which reissues a request with probability q after a delay d. We show how the incorporation of randomness within SingleR policies enables requests to be reissued earlier while still meeting a specified reissue budget. This allows for SingleR to reduce tail-latency significantly even when constrained by a small reissue budget.

This section is organized as follows. Section 8.2.1 presents the model and terminology. Section 8.2.2 defines the ***Single-Time / Deterministic*** (SingleD) policies which formalize the "delayed reissue" strategy of prior work. We present SingleR policies in Section 8.2.3 and discuss their benefits over SingleD in Section 8.2.4.

8.2.1 Model and terminology

We shall, for the moment, operate within a simplified performance model in which there are no queueing delays and query response-times are independent and identically distributed. Later, in Section 8.4.2 we describe how these limitations can be overcome to adapt our techniques to workloads with correlated response-times and queueing delays.

Formally, we consider an interactive workload to be a collection of queries where each query is composed of exactly one ***primary request*** that is dispatched at time $t = 0$ and zero or more ***reissue requests*** dispatched at times $d \geq t$.

The response-time of a query is based on the length of time between the dispatch of the primary request and the arrival of any reply from either a primary or reissue request.

The ***reissue rate*** of a workload consisting of N queries and M reissue requests is defined as the ratio M/N.

We look for a reissue policy that minimizes a workload's kth percentile tail-latency with the reissue rate equal to a given ***reissue budget*** B.

8.2.2 The SingleD policies

The ***Single-Time / Deterministic*** (SingleD) policy family is a 1-parameter family of policies that is parametrized by a ***reissue delay*** d. A SingleD policy reissues a request if a response has not been received after d seconds.

Let the random variable X denote the response time of the primary request and Y denote the response time of the reissue request. A query Q completes before time t if its primary response-time X is less than t, or if the reissue request response-time Y is less than $t - d$. The probability that the query Q responds before time t is given by Equation (8.1).

$$\Pr(Q \leq t) = \Pr(X \leq t) + \Pr(X > t)\Pr(Y \leq t - d) \tag{8.1}$$

The expected number of reissue requests created by a SingleD policy is equal to the number of primary requests that respond after time d, i.e., the reissue budget is

$$B = \Pr(X > d)\ . \tag{8.2}$$

Therefore, if a system can tolerate 10% additional requests, then the delay d is chosen for the SingleD policy such that $\Pr(X > d) = 1/10$. The smaller the delay d, more requests are reissued, and the higher the budget B.

8.2.3 The SingleR policies

The ***Single-Time / Random*** (SingleR) policy family is a 2-parameter family of policies that is parametrized by a ***reissue delay*** d and a ***reissue probability*** q. A SingleR policy reissues a request with probability q if a response has not been received after d seconds.

A query Q responds before time t if the primary request responds before time t, or if a reissue request was created and its response time is less than $t - d$. The probability that Q completes before time t while employing SingleR is given by Equation (8.3).

$$\Pr(Q \leq t) = \Pr(X \leq t) + q \cdot \Pr(X > t)\Pr(Y \leq t - d) \tag{8.3}$$

The reissue budget is

$$B = q \cdot \Pr(X > d) \tag{8.4}$$

Given Equation (8.3) and Equation (8.4), we write the constrained optimization problem which identifies the reissue delay and probability parameters of the optimal SingleR policy given the primary and reissue response time distributions X and Y.

Optimal policy for SingleR

Given tail-latency percentile k, a reissue budget B, and policy family SingleR

$$\begin{aligned} \underset{d,\ q}{\text{minimize}} \quad & t \\ \text{subject to} \quad & \Pr(X \leq t) + q \cdot \Pr(X > t)\Pr(Y \leq t - d) \geq k, \\ & q \cdot \Pr(X \geq d) \leq B \end{aligned}$$

8.2.4 Randomization is essential

The use of randomization in SingleR allows the reissue budget, and thus the added resource and system load, to be bounded while also ensuring that requests can be reissued early enough so they have sufficient time to respond. This may not be allowed under SingleD, which we illustrate in the following example.

Suppose, for example, that we want to minimize a workload's 95th percentile tail-latency by reissuing no more than 5% of all queries. Clearly, this cannot be achieved using a SingleD policy — its limited reissue budget forces it to reissue requests later than the original 95th percentile tail-latency.

In general, a SingleD policy cannot reduce *any* workload's kth percentile latency with budget $B < 1 - k$. Randomization is an essential part of an effective reissue policy.

8.3 Single versus multiple reissue

As we saw in Section 8.2, randomness provides SingleR policies an important degree of freedom that enables a continuous trade-off between the advantages of immediate and delayed reissuing. A natural question arises: can we obtain an even better policy family by introducing additional degrees of freedom?

In this section, we address this question by introducing MultipleR policies that can reissue requests more than once, at multiple different times, and with different probabilities. We prove a surprising fact: for a given reissue budget B and tail-latency percentile k, the optimal MultipleR and SingleR policies achieve the same tail-latency reduction.

Note that we continue to operate in the simplified model described in Section 8.2.1 in which there are no queueing delays and query response-times are independent and identically distributed. These limitations will be lifted in Section 8.4.2 as we show how to adapt SingleR policies to handle correlated response-times and queueing delays.

8.3.1 Multiple time policies

The ***Multiple-Time / Random*** (MultipleR) policy family contains policies that can reissue requests multiple times. A policy that reissues a request at-most n times consists of a sequence of n delays $d_1, d_2, \ldots, d_n$ and n probabilities $q_1, q_2, \ldots, q_n$. Like SingleR, the MultipleR family explores the space between two extremes — the "immediate reissue" and "delayed reissue" strategies. Specifically, the reissue times d_i of a MultipleR policy lie between 0, the time of immediate reissue, and d', the time selected by a "delayed reissue" SingleD policy, where $\Pr(X > d') = B$. For any d_i, since $d_i \leq d'$, the following condition holds

$$\Pr(X > d') \geq B \ . \tag{8.5}$$

For the purpose of our later arguments, we also define the ***Double-Time / Random*** (DoubleR) policy family. The DoubleR family is a subset of MultipleR and contains policies that reissue requests at most twice.

8.3.2 Single is optimal

We prove the optimality of SingleR in two steps: (1) We show in Theorem 44 that the optimal policies in the SingleR and DoubleR families achieve identical tail-latency reduction; (2) Finally, we prove a generalization in Theorem 45 for MultipleR policies that have $n > 2$ reissue times.

Theorem 44 *The optimal* SingleR *and* DoubleR *reissue policies achieve the same kth percentile tail-latency when given the same reissue budget B.*

PROOF. Consider the optimal SINGLER policy with budget B that minimizes t, the kth percentile tail-latency. Suppose that this policy reissues requests at time d^*. Then, the probability that a query using the optimal SINGLER policy responds before time t is given by Equation (8.6) below.

$$\Pr(Q \leq t) = \Pr(X \leq t) + G^*_{SR} \tag{8.6}$$

where,

$$G^*_{SR} = \frac{B}{\Pr(X > d^*)} \Pr(X > t) \Pr(Y \leq t - d^*) \ . \tag{8.7}$$

The first term $\Pr(X \leq t)$ is the probabilitity that the primary request returns before the tail-latency deadline. The term G^*_{SR} corresponds to the case for which the primary request misses the deadline, but the reissue request responds on-time.

Now consider a DOUBLER policy with reissue times d_1, d_2 and reissue probabilities q_1, q_2. The probability that a query using this policy reponds before time t is given by Equation (8.8) below.

$$\Pr(Q \leq t) = \Pr(X \leq t) + G_1 + G_2 \tag{8.8}$$

where,

$$G_1 = q_1 \Pr(X{>}t) \Pr(Y_1{\leq}t - d_1) \tag{8.9}$$

$$G_2 = q_2(1 - q_1 \Pr(Y_1{\leq}t - d_1)) \Pr(X{>}t) \Pr(Y_2{\leq}t - d_2) \tag{8.10}$$

The term G_1 corresponds to the case for which the primary request misses the deadline, but the first reissue request responds on-time. Lastly, the third term G_2 corresponds to the case where both the primary and first reissue request miss the deadline, but the second reissue request responds on-time.

We shall show that $G_1 + G_2 \leq G^*_{SR}$. After this has been shown, it follows that no DOUBLER policy can achieve a lower tail-latency than a SINGLER policy with the same budget.

First, we provide a bound on G_1.

Consider a SINGLER policy that reissues requests at time d_1 with probability $B \cdot \Pr(X > d_1)^{-1}$. Using this policy, the probability that a query returns before time t is given by

$$\Pr(Q \leq t) = \Pr(X \leq t) + G_{SR,1} \tag{8.11}$$

where,

$$G_{SR,1} = \frac{B}{\Pr(X > d_1)} \Pr(X > t) \Pr(Y \leq t - d_1) \ . \tag{8.12}$$

Since G^*_{SR} is the optimal policy for a budget B, we have that

$$G_{SR,1} \leq G^*_{SR} \ . \tag{8.13}$$

Multiplying both sides of Inequality (8.13) by $q_1 \Pr(X > d_1) B^{-1}$ gives us the upper bound on G_1 shown in Inequality (8.14).

$$G_1 \leq \frac{q_1 \Pr(X > d_1)}{B} G^*_{SR} \tag{8.14}$$

Second, we provide an upper bound on G_2.

We begin by formulating an upper bound on G_2 that is a function of q_1. This requires a sequence of observations. We note that the budget constraint for the DOUBLER policy implies the following inequality:

$$q_1\Pr(X > d_1) + q_2\Pr(X > d_2)(1 - q_1\Pr(Y_1 \le d_2 - d_1)) \le B \tag{8.15}$$

Then, given q_1 Inequality (8.15) implies the following upper bound on q_2:

$$q_2 \le \frac{B - q_1\Pr(X > d_1)}{\Pr(X > d_2)(1 - q_1\Pr(Y_1 \le d_2 - d_1))} \ . \tag{8.16}$$

Finally, we incorporate this bound on q_2 into the expression for G_2 given in Equation (8.10) to obtain an upper bound on G_2 as a function of q_1.

$$G_2 \le \frac{B - q_1\Pr(X > d_1)}{\Pr(X > d_2)} \gamma \ \Pr(X > t)\Pr(Y_2 \le t - d_2) \tag{8.17}$$

where, $\gamma = (1 - q_1\Pr(Y_1 \le t - d_1))/(1 - q_1\Pr(Y_1 \le d_2 - d_1))$. Note that γ is at most 1 since d_2 is less than t which allows us to omit γ in Inequality (8.17) to obtain a simpler (albeit weaker) upper bound on G_2.

Now consider a SINGLER policy that reissues at time d_2 with probability $B\Pr(X > d_2)^{-1}$. The probability that a query using this policy responds before time t is given by:

$$\Pr(Q \le t) = \Pr(X \le t) + G_{SR,2} \tag{8.18}$$

where,

$$G_{SR,2} = \frac{B}{\Pr(X > d_2)}\Pr(X > t)\Pr(Y_2 \le t - d_2) \ . \tag{8.19}$$

We have that for all positive a that $aG_{SR,2} \le aG^*_{SR}$. Let $a = 1 - q_1\Pr(X > d_1)B^{-1}$, which is strictly positive since the budget constraint on the DOUBLER policy implies the inequality $q_1\Pr(X > d_1) < B$.

Then, combining Equation (8.19) and Inequality (8.17) we have that

$$\begin{aligned} G_2 &\le \left(1 - \frac{q_1\Pr(X > d_1)}{B}\right) G_{SR,2} \\ &\le \left(1 - \frac{q_1\Pr(X > d_1)}{B}\right) G^*_{SR} \ . \end{aligned} \tag{8.20}$$

Together the upper bounds on G_1 and G_2 imply that $G_1 + G_2 \le G^*_{SR}$, completing the proof. □

Theorem 45 *The optimal* SINGLER *and* MULTIPLER *reissue policies achieve the same kth percentile tail-latency when given the same reissue budget B.*

PROOF.

Assume as an inductive hypothesis that the theorem holds for n- and $(n + 1)$-time MULTIPLER policies. The base cases for 1-time and 2-time MULTIPLER policies follows from Theorem 44.

Consider an optimal $(n+2)$-time MULTIPLER policy P_{n+2} with reissue times $d_1, \ldots, d_{n+2}$. To complete the inductive argument, we will show that there exists an $(n + 1)$-time MULTIPLER policy with reissue times $d_1, \ldots, d_n, d'$ that achieves the same kth percentile tail-latency.

Let P_n be the n-time MultipleR policy obtained by taking the first n reissue times and reissue probabilities in P_{n+2}. The policy P_n consumes budget $\alpha B (\leq B)$, where $\alpha \leq 1$.

Let $Q[P_n]$ be a random variable representing the response-time distribution of a query reissued using policy P_n.

Let's now transform the original problem to a new but equivalent problem of minimizing the kth percentile tail-latency of a workload W' with primary response-time distribution $Q[P_n]$ and reissue response-time distribution Y.

We want to show that, for the workload W', a reissue policy with budget $(1-\alpha)B$ that reissues at times d_{n+1} and d_{n+2} is a DoubleR policy. In particular, we want to show that its budget and reissue times satisfy the condition of Inequality (8.5) under MultipleR definition, i.e., the following two inequalities hold:

$$\Pr(Q[P_n] \geq d_{n+1}) \geq (1-\alpha)B \tag{8.21}$$

$$\Pr(Q[P_n] \geq d_{n+2}) \geq (1-\alpha)B \tag{8.22}$$

In order to show that Inequality (8.21) and Inequality (8.22) hold, we use the induction hypothesis for n-time MultipleR policies to obtain a lower-bound on $\Pr(Q[P_n] \geq d_{n+1})$ and $\Pr(Q[P_n] \geq d_{n+2})$.

Let $k' = (1 - \Pr(Q[P_n] > d_{n+1}))$ so that d_{n+1} is the k'th percentile tail-latency of $Q[P_n]$. Consider the original workload W with primary response-time X and reissue response-time Y. By the induction hypothesis for n-time MultipleR policies, there exists a SingleR policy P_{SR} with budget αB that achieves a k'th percentile tail-latency that is at most d_{n+1}. Suppose that P_{SR} reissues requests at time d^*. Then, we have that

$$\Pr(Q[P_n] > d_{n+1}) \geq \Pr(Q[P_{SR}] > d_{n+1}) \tag{8.23}$$

and that

$$\frac{\Pr(Q[P_{SR}] > d_{n+1})}{\Pr(X > d_{n+1})} = 1 - \frac{\alpha B \Pr(Y \leq d_{n+1} - d^*)}{\Pr(X > d^*)} \tag{8.24}$$

By the definition of MultipleR we have that $\Pr(X > d_{n+1}) \geq B$ and by the definition of SingleR that $\Pr(X > d^*) \geq B$. Together with Inequality (8.23) this implies that

$$\Pr(Q[P_n] > d_{n+1}) \geq \Pr(Q[P_{SR}] > d_{n+1}) \geq (1-\alpha)B \tag{8.25}$$

Which proves that Inequality (8.21) holds. The proof that Inequality (8.22) holds follows an identical argument.

Therefore, we have shown that for the workload W' the policy which reissues requests at times d_{n+1} and d_{n+2} is a DoubleR policy. By Theorem 44 it follows that there exists a SingleR policy that reissues at some time d' which achieves the same kth percentile tail-latency as this DoubleR policy. We can, therefore, replace the $(n+2)$-time MultipleR policy with an $(n+1)$-time MultipleR policy that reissues at times $d_1, \ldots, d_n, d'$ that achieves the same kth percentile tail-latency — completing the proof. □

Analysis with Correlation The analysis in Theorem 44 may be extended (with additional assumptions) to the case in which primary and reissue response times are correlated. Consider a DoubleR policy that reissues requests at times d_1, d_2, and let Q_1 represent the probability that either the primary or first reissue request (issued at time d_1) responds before time t. Then the analysis in Theorem 44 holds if a) $\Pr(Y_2 \leq t - d_2 | Q_1 > t) \leq$

$\Pr(Y_2 \leq t - d_2 | X > t)$, and b) $\Pr(Y_1 \leq d_2 - d_1) | X > d_2) \leq \Pr(Y_1 \leq t - d_1 | X > t)$. The first assumption (a) is fairly modest and is employed to simplify Inequality (8.15). Intuitively, assumption (a) states that the likelihood of a second reissue request responding before time $t - d_2$ decreases (or is unchanged) if the first reissued request fails. The second assumption (b) is a technical requirement that allows our proof to use the budget constraint in Inequality (8.15) in the correlated case. Specifically, assumption (b) ensures that γ in Inequality (8.17) is at most 1. Informally, assumption (b) states that the positive correlation between primary and reissue response-times is weaker in the tail of the distribution (i.e. near time t) than it is near the reissue times d_1, d_2. We note that in the case where assumption (b) fails to hold, derived bounds on γ can still be used to obtain competitive ratios.

The optimality of SingleR is a powerful result, restraining the complexity of reissue policies to one time reissue only while guaranteeing its effectiveness.

8.4 SingleR for interactive services

This section presents how to use SingleR for interactive services: We use a data-driven approach to efficiently find the appropriate parameters, reissue time and probability, given sampled response times of the workloads. We develop the parameter search algorithm in 3 steps. (1) We start from a simple model in Section 8.4.1, assuming the response times of primary and reissue requests are independent. We present an algorithm ComputeOptimalSingleR that computes optimal reissue time and probability, minimizing tail latency. Our algorithm is computationally efficient, taking $O(N \log N)$ time where N is the number of response time samples. (2) We extend the algorithm in Section 8.4.2 to incorporate correlation between reissue and primary requests, guaranteeing optimality on parameter selection while offering the same computational efficiency of $O(N \log N)$. (3) We show how to adaptively refine a SingleR policy to take into account additional queueing delays introduced to the system by the reissue requests in Section 8.4.3.

8.4.1 Parameter search

The ComputeOptimalSingleR(R_X, R_Y, k, B) procedure (in Figure 8-1) computes the optimal SingleR policy to minimize the kth percentile tail-latency of an interactive service with reissue budget B. The response-time distributions for the service are represented using two sets of samples: a set R_X of response times for primary requests; and, a set R_Y of response times for reissued requests, accommodating the cases in which these distributions differ, e.g., when reissue requests are executed using dedicated or specialized resources. The output of the procedure is the reissue time d^* and the reissue probability q for the SingleR policy.

ComputeOptimalSingleR searches for the optimal reissue time. We preserve the following invariant throughout the procedure — the SingleR policy that reissues requests at time d^* achieves a kth percentile tail-latency of at most t. The procedure begins on lines 2–3 by selecting a trivial policy that reissues all requests at time $d^* \leftarrow \min\{R_X\}$ and achieves a kth percentile tail-latency of $t \leftarrow \max\{R_X\}$. A search is then performed on lines 4–12 for each reissue time $d \in R_X$ to determine if the SingleR policy reissuing at time d achieves a kth percentile tail-latency smaller than t. For each time d, the success-rate α of the SingleR policy that reissues at time d is computed on line 7, which is the probability that a query is serviced before time t. If this success rate is greater than the tail-latency

```
ComputeOptimalSingleR($R_X$, $R_Y$, $k$, $B$):
 1  $Q \leftarrow R_X$
 2  $d^* \leftarrow \min\{Q\}$
 3  $t \leftarrow \max\{Q\}$
 4  while $Q \neq \emptyset$
 5      $d \leftarrow \min\{Q\}$
 6      $Q \leftarrow Q - \{d\}$
 7      $\alpha \leftarrow$ SingleRSuccessRate($R_X, R_Y, B, t, d$)
 8      while $\alpha > k$ and $t > d$
 9          $Q \leftarrow Q - \{t\}$
10          $t \leftarrow \max\{Q\}$
11          $d^* \leftarrow d$
12          $\alpha \leftarrow$ SingleRSuccessRate($R_X, R_Y, B, t, d$)
13  $q \leftarrow 1 -$ DiscreteCDF($R_X, d^*$)
14  return $(d^*, q)$
```

```
SingleRSuccessRate($R_X$, $R_Y$, $B$, $t$, $d$):
1  $\Pr(X \leq t) \leftarrow$ DiscreteCDF($R_X, t$)
2  $\Pr(X > d) \leftarrow 1 -$ DiscreteCDF($R_X, d$)
3  $\Pr(Y \leq t - d) \leftarrow$ DiscreteCDF($R_Y, t - d$)
4  $q \leftarrow B / \Pr(X > d)$
5  $\alpha \leftarrow \Pr(X \leq t) + q \cdot (1 - \Pr(X \leq t)) \cdot \Pr(Y \leq t - d)$
6  return $\alpha$
```

```
DiscreteCDF($R$, $t$):
1  $s \leftarrow |\{x \in R;\ x < t\}|$
2  return $s/|R|$
```

Figure 8-1: Pseudocode for the data-driven algorithm for finding the optimal SingleR policy.

percentile target k, we replace d^* with $d^* \leftarrow d$ and decrease t to $\max\{R_X - \{t\}\}$ while preserving the invariant. This iterative refinement of the policy is repeated on lines 8–12 until the success rate α of the SingleR policy reissuing at time d is less than k. By then, we find the optimal d^* value, and its corresponding q value is computed at line 13.

Complexity.

ComputeOptimalSingleR is computationally efficient with complexity of $\Theta(N + \mathit{Sort}(N))$ where N is the number of samples, and $\mathit{Sort}(N)$ is the time required to sort N response times. In particular, the list of potential reissue times Q is initialized with N response times. Each time SingleRSuccessRate is invoked one element is removed from Q. Therefore, SingleRSuccessRate can be invoked at most N times. SingleRSuccessRate evaluates three cumulative distribution functions DiscreteCDF on lines 1–3. Although the success rate α computed on line 5 is not necessarily monotonic as a function of (t, d), its composite CDFs are monotonic in t, d, and $t - d$ respectively. As a result, the amortised cost of DiscreteCDF is $O(1)$ with a careful analysis considering order statistics and using finger search tree [53, 141]. DiscreteCDF takes pre-sorted response time samples as inputs, where the sorting takes $\Theta(\mathit{Sort}(N))$ time. Summing them together, the complexity of ComputeOptimalSingleR is $\Theta(N + \mathit{Sort}(N))$.

Theorem 46 shows that ComputeOptimalSingleR is a computationally-efficient linear time algorithm.

Theorem 46 *Given N sorted response-times R_X and N sorted reissue response-times R_Y,* ComputeOptimalSingleR *computes the optimal* SingleR *policy for the case in which request response-times are independent in $\Theta(N)$ time.*

Proof.

The list of potential reissue times Q is initialized with N response times. Each time SingleRSuccessRate is invoked one element is removed from Q. Therefore, SingleRSuccessRate can be invoked at most N times.

The SingleRSuccessRate procedure evaluates three cumulative distribution functions on lines 1–3. The runtime of DiscreteCDF depends on the time required to perform the order statistic query on line 1. These order statistic queries can be evaluated in $O(1)$ amortized time by taking advantage of the temporal access pattern of ComputeOptimalSingleR. The algorithm considers N reissue times $d = d_1, d_2, \ldots, d_N$ on lines 4–12 in ascending order, and the tail-latency target t monotonically decreases as better parameters are discovered. These two properties imply that $t - d$ is also monotonically decreasing. The monotonic behavior of t, d, and $t - d$ is exploited by storing the response times R_X and R_Y in a finger search tree [53, 141, 165, 198, 240] that supports consecutive searches to tree elements whose order statistic differs by c in $O(\lg c)$ time. This property implies that a sequence of N order-statistic queries for monotonic sequences can be performed in $\Theta(N)$ time. □

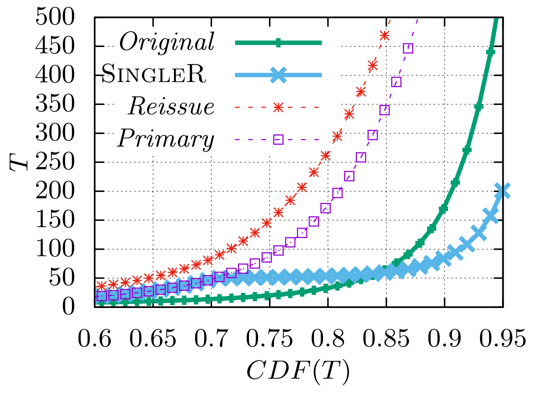
8.4.2 Incorporating response-time correlations

The response-time of a request can be divided into two components: the amount of time a request waits in a server's queue before being processed (the ***queueing time***), and the time required execute the request (the ***service time***). The response-times of primary and reissue requests, however, will often be correlated. For example, queries within a workload can have different service times: a query with high service time (e.g., many instructions) is likely to take long for both primary and reissue requests. The system's instantaneous load may be similar upon the arrival of the primary and reissue requests.

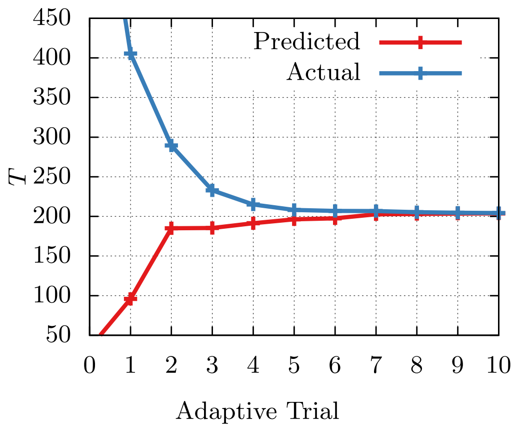
Correlations between primary and reissue requests influence the probability that a reissue request will respond before a tail-latency deadline. This influence can be taken into account in ComputeOptimalSingleR by modifying line 5 of SingleRSuccessRate in Figure 8-1 to use the conditional distribution $\Pr(Y \leq t-d | X > t)$ in place of $\Pr(Y \leq t-d)$.

The conditional distribution $\Pr(Y \leq t - d | X > t)$ may be estimated efficiently by using a $2D$ orthogonal range query data structure [238, 2] over pairs (t_x, t_y) where t_x and t_y are the primary and reissue response times.

Each range query performed within SingleRSuccessRate takes $O(\log N)$ time, and SingleRSuccessRate is invoked at most $2N$ times by ComputeOptimalSingleR. Therefore, the procedure ComputeOptimalSingleR which takes into account correlation computes the optimal SingleR policy in $\Theta(N \lg N)$ time.



(a) Inverse CDF.



(b) Adaptive algorithm.

Figure 8-2: Convergence of the adaptive SingleR policy on a workload with correlated service-times and queueing delays.

8.4.3 Iterative adaptation for queue delays

The queueing delay of requests in a workload depends on the arrival process to a service. The use of a reissue policy can perturb this arrival process and change the response-time distributions used by ComputeOptimalSingleR to find a SingleR policy.

The impact of added load on a workload's response-time distributions can be significant. Consider the inverse CDFs illustrated in Figure 8-2a for *Original* and *Primary* requests[1]. The *Original* curve illustrates the inverse CDF of the original primary response-time distribution of the system when no requests are reissued. The *Primary* curve illustrates the new inverse CDF of the primary response-time distribution when using a SingleR policy with a 30% reissue budget. The impact of these reissue requests on the primary response-time distribution is dramatic: the 85th percentile grows from 50 to 350.

We employ an adaptive approach to iteratively refine a SingleR policy in-response to changes in the response-time distribution. First, we begin with a reissue policy P that reissues requests at time $d = 0$ with probability B. We then execute the system with the reissue policy and sample the response-time distributions of primary and reissue requests. The sampled response-time distributions are used within ComputeOptimalSingleR to compute the optimal SingleR policy P_{local} for these response-time distributions. Next, we obtain a new policy P' that has reissue delay $d' = d + \lambda(d_{local} - d)$ where λ is a learning rate. Finally, this process is repeated until the empirical kth percentile tail-latency converges to the value predicted by ComputeOptimalSingleR and the empirical reissue rate converges to B.

This adaptive approach is based upon two observations: a) using the same budget, reissuing later tend to impact load more as it is more likely to reissue requests with more work and higher resource demands; and, b) small changes to the reissue delay result in only small changes to the response-time distributions. Observation (a) implies that the predicted kth percentile tail-latency at each step of ComputeOptimalSingleR increases after each step of the algorithm. Observation (b) implies that for sufficiently small λ that the true optimal reissue time d_{opt} lies between d' and d_{local} at each step of the algorithm.

Figure 8-2b shows the 95th percentile tail-latency achieved on each step of the adaptive

[1]The corresponding simulation setup for Figure 8-2a is discussed in Section 8.5.

algorithm using a learning rate of 0.2 for a SinglеR policy with a reissue budget of 30%. Convergence can be detected by comparing the policy optimizer's predicted tail-latency with the observed latency when using the policy. For this workload, convergence is achieved after ≈ 6 iterations.

8.4.4 Extended scenarios

The tools and algorithms presented in the preceding sections can be applied to handle common scenarios that occur in practice. Since space limitations prohibit an exhaustive examination of each of these scenarios, we shall instead sketch a few strategies for addressing common use cases.

Varying load / response-time distributions.

In practice a system's response-time distribution can vary over time on both short (hourly, daily), and long (monthly, yearly) time scales. The iterative algorithm for adaptively refining a SingleR policy can be applied in an online fashion to address these temporal variations, but requires modifications which depend on specific application needs and the time-scale of interest to properly balance exploration and exploitation in its search.

Selecting optimal reissue budget.

The adaptive algorithm described in this section assumes the use of a fixed reissue budget. As we learned in Section 8.2, SingleR policies are able to reduce tail-latency in a "smooth" fashion even with very small reissue budgets. As a consequence, the tail-latency reduction of SingleR as a function of the reissue budget tends to be a parabola whose extrema can be readily found through simple binary search techniques.

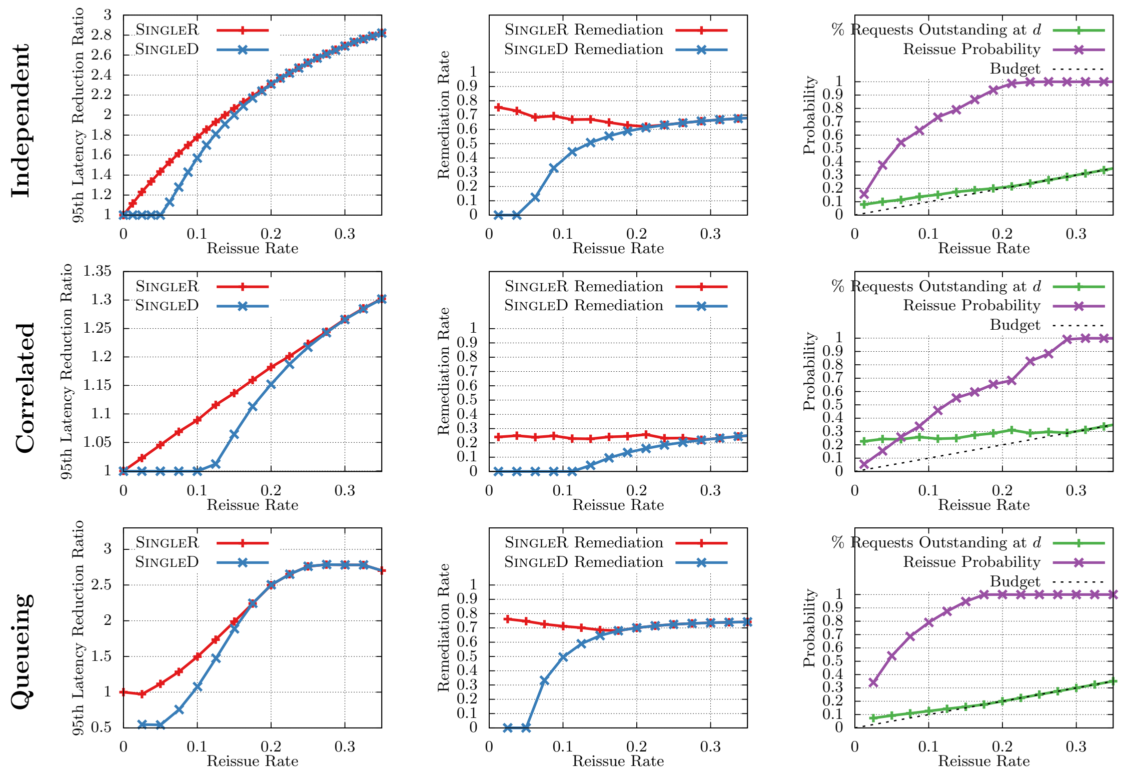
To evaluate the practicality of this simple approach, we implemented a simple budget selection procedure that performs the following steps: 1) set $\delta = 1\%$ and set *best-budget* $= 0$; 2) for budget *best-budget* $+ \delta$ run the adaptive SingleR policy optimizer for 5 adaptive trials to produce reissue policy P; 3) collect response-time data from the system when using reissue policy P; 4) if the budget *best-budget* $+ \delta$ has smaller 99th percentile tail-latency than *best-budget*, then set $\delta = 3\delta/2$. Otherwise, set $\delta = -\delta/2$. An example of this binary search procedure is presented later in Figure 8-8 as part of our system experiments in Section 8.6.

Meeting tail-latency with minimal resources

Interactive services often formulate service-level agreements (SLA) that guarantee a fixed latency for $k\%$ of all requests. In such a scenario, a system designer may be interested in minimizing the resources required to satisfy the SLA. Given a particular tail-latency target T, the budget can be minimized using either a brute force search, starting at small reissue rates, or by using a variation of the binary search procedure for finding the optimal budget that transforms tail-latency values L using the function $f(L) = \min\{T, L\}$.

8.5 Simulations

In this section we use a discrete-time event simulator to carefully evaluate the behavior and tail-latency impact of SingleR policies. Simulation allows us to vary workload and system



(a) Tail-latency reduction ratio.

(b) Reissue remediation rate.

(c) Reissue time and probability.

Figure 8-3: Simulation results for SingleR and SingleD policies with varied reissue budgets on three simulated workloads: *Independent*, *Correlated*, and *Queueing*.

properties covering a wide range of scenarios.

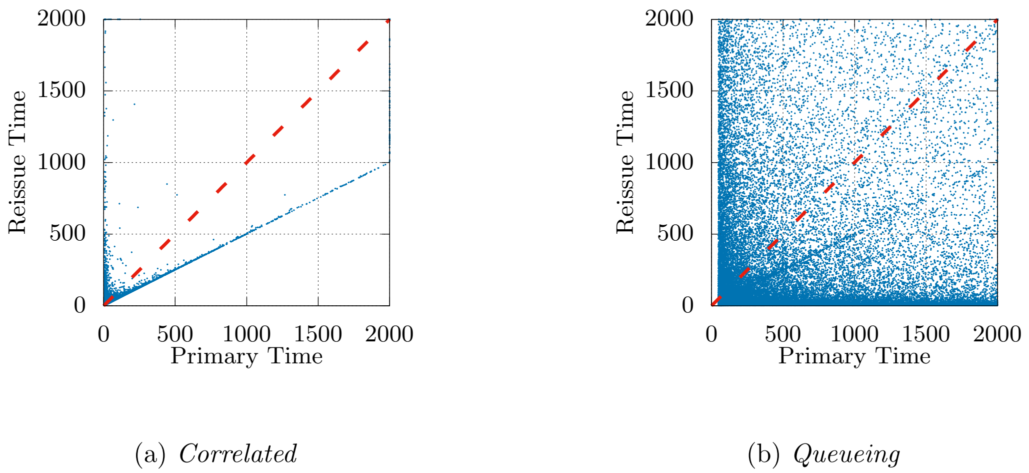
First, we provide simulation results on three types of workloads: *Independent*, *Correlated*, and *Queueing*, corresponding to the three workload models in Section 8.4. This experiment demonstrates two points: a) Randomness in SingleR is, in fact, especially important for workloads with correlated service-times and queueing delays; and, b) The optimal SingleR policy takes workload characteristics into account in order to maximize the value of each reissued request.

Next, we conduct a sensitivity study that varies the *Queueing* workload along many dimensions: utilization, service time distribution, percentile targets, strength of service-time correlations, load balancing strategies, and request prioritization strategies. The results demonstrate SingleR is effective and robust over varying workloads and system design properties.

8.5.1 Simulated workload

Figure 8-3 provides simulation results on a set of three workloads: *Independent*, *Correlated*, and *Queueing*. The service-times in each workload are drawn from a Pareto distribution with shape parameter 1.1 and mode 2.0.

In the *Independent* workload, the service-times of primary and reissue requests are independent and have no queueing delays (i.e. there are an infinite number of servers). In the *Correlated* workload, the primary and reissue request service-times are correlated via



(a) *Correlated* (b) *Queueing*

Figure 8-4: Response-time correlations between primary and reissue requests on the *Correlated* and *Queueing* workloads. The service-time X of the primary request is drawn from a Pareto distribution with shape 1.1 and mode 2. The service-time Y of a reissued request is drawn from $Y = rx + Z$, where x is the observed service-time of the primary request, Z is drawn from a Pareto distribution with shape 1.1 and mode 2, and $r = 0.5$.

the relationship $Y = rx + Z$ where x is the sampled primary request service-time, Z is an independently drawn service-time, and $r = 0.5$ is a linear correlation ratio. In the *Queueing* workload, requests have correlated service-times and arrive according to a Poisson process. The request is dispatched to the FIFO queue of one of 10 servers selected uniformly at random. The arrival rate is chosen to achieve a system utilization of 30%.

Figure 8-3a compares the 95th percentile tail-latency reduction achieved by the optimal SingleR and SingleD policies for varied reissue budgets. For the *Queueing* workload, both the SingleR and SingleD policies are selected using adaptive policy refinement (for the SingleD policy this adaptive refinement is needed to ensure the reissue budget is satisfied). Figure 8-3b illustrates the "remediation rate" of SingleR and SingleD policies. The ***remediation rate*** measures the average value of added (i.e. actually issued) reissue requests and is defined to be the probability that a primary request X exceeds a tail-latency target t, but the reissued request Y responds before time $t - d$, i.e. $\Pr(X > t \cap Y < t - d)$. Figure 8-3c plots the reissue times and probabilities used by the optimal SingleR policy for each budget.

8.5.2 Benefits of randomization

The results of Figure 8-3a illustrates the benefits of randomization in reissue policies. For all three workloads, there exists a range of reissue budgets for which the SingleD policy is ineffective at reducing the 95th percentile tail-latency. On the *Independent* workload a SingleD policy is unable to achieve any tail-latency reduction when the reissue budget is less than 5%. On the *Correlated* workload, SingleD policies are ineffective for reissue budgets less than 10%. Worst of all, SingleD policies actually *increases* the 95th percentile latency of the *Queueing* workload with reissue budgets less than 10% — since these reissued requests increase system load.

In contrast, SingleR is able to reduce the 95th percentile tail-latency for all reissue budgets on the *Independent* and *Correlated* workloads. On the *Queueing* workload, SingleR begins to reduce tail-latency once the reissue budget is greater than 3%. For all three workloads, randomization allows for SingleR to achieve better tail-latency reduction than SingleD for budgets less than 15%.

8.5.3 Impact of correlation and queueing

The procedure outlined in Section 8.4 for finding an optimal SingleR policy takes into account the properties of the primary and reissue response-time distributions, and adapts to queueing delays. By inspecting the three workloads in Figure 8-3, we can gain insight into how SingleR reissue policies are able to outperform SingleD.

The goal of ComputeOptimalSingleR is to find a SingleR policy that minimizes the workload's kth percentile tail-latency with a reissue budget of B. One can think of ComputeOptimalSingleR as searching over all policies that use budget B in order to find the policy which maximizes the value of each added request. A convenient measure of the "value" of each reissue request is its remediation rate — i.e. the probability that the redundancy provided by the reissue request was necessary for the query to meet its tail-latency target.

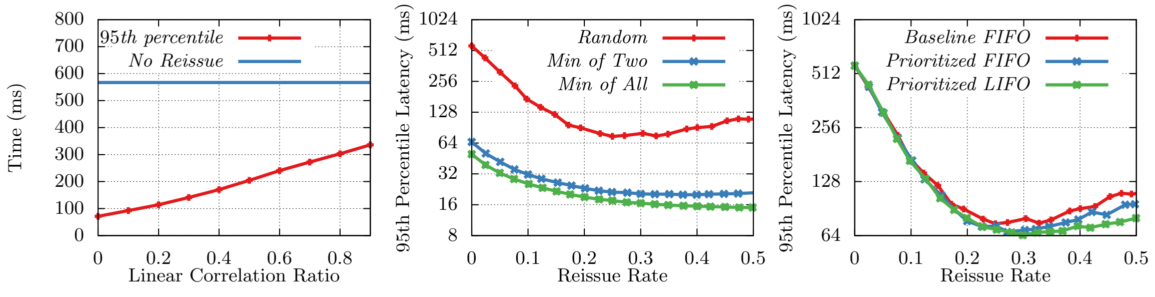
Figure 8-3c illustrates the way in which SingleR changes its choice of reissue delay and probability based upon the reissue budget and workload characteristics. We shall discuss the behavior of SingleR policies for each of our three workloads in the case where the reissue budget is 10%.

On the *Independent* workload, the optimal SingleR policy reissues requests with probability 0.7 at a time d where approximately 15% primary requests remain outstanding — resulting in approximately 10% of all requests being reissued in total. On the *Correlated* workload, the optimal SingleR policy chooses to reissue requests with probability 0.4 at a time d where 25% of requests are outstanding.

On the *Correlated* workload, the optimal SingleR policy must reissue requests earlier due to service-time correlations. When optimizing its success rate it takes into account the fact that if a query's primary request exceeds a tail latency target, there is a higher chance of its reissue request responding slowly. By reissuing requests earlier in time, the probability that the reissued request will help tail latency (i.e. the remediation rate) increases. Therefore, on this workload the optimal policy reissues requests earlier at a time d when 40% of requests are outstanding, and reissues with a smaller probability of 25%.

On the *Queueing* workload, the optimal SingleR policy reissues requests with probability 0.8 at a time d where approximately 13% of requests are outstanding. Although this workload's service-times are correlated, the latency of requests in the tail of the response-time distribution is dominated by queueing delays which depends on the service process as well as on request arrival process and load balancing. Indeed, we can observe in Figure 8-4b that the addition of queueing delays dampens the strength of correlation between primary and reissue requests. Although the preexisting correlation can still be observed, the structure of the joint-distribution exhibits more randomness due to request queueing time. This provides an explanation for why SingleR, and reissuing in general, can achieve more latency reduction on the Queueing workload than the Correlated workload, as shown in Figure 8-3a.

This experiment shows that SingleR optimizes the choice of reissue time and probability based upon workload characteristics, maximizing the benefit of reissued requests for



(a) Correlation (b) Load-balancing (c) Queuing

Figure 8-5: Illustrates impact of correlation ratio (Figure 8-5a), load-balancing strategies (Figure 8-5b), and server's queue-management policies (Figure 8-5c) on the 95th percentile tail-latency of the *Queueing* workload.

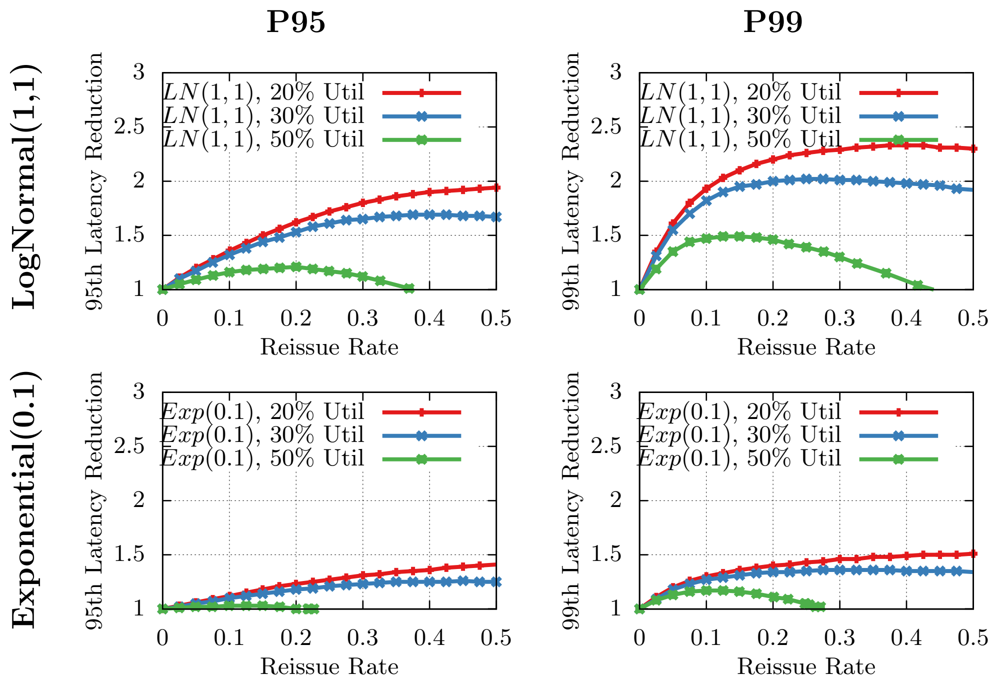


Figure 8-6: The 95th percentile tail-latency $P95$ and the 99th percentile tail-latency $P99$ for Exp(0.1) and LogNormal(1, 1) distributions with varied reissue rates and system utilization.

tail-latency reduction.

8.5.4 Sensitivity study

We study the sensitivity of SINGLER to workload properties and design choices, including: utilization, service-time distribution, target latency percentiles, correlation among requests, load-balancing among servers, and changing the priority processing reissued requests. As a baseline, we use a variant of the Queueing workload from Section 8.5.1 without service-time correlations unless otherwise specified.

Utilization, service-time distribution and percentiles

We use LogNormal$(1, 1)$ and Exponential(0.1) as service time distributions and measure P95 and P99 tail latency reduction for three utilization levels: 20%, 30%, and 50%.

Figure 8-6 illustrates the $P95$ and $P99$ tail-latency reduction (Y-axis) achieved by SingleR policies over a range of reissue budgets (X-axis) compared to the original tail-latencies when no requests are reissued. The results demonstrate: (1) Reissue obtains higher benefit under less loaded systems, but even at rather high load of 50% utilization, SingleR achieves latency reduction of up to 1.5 times. (2) The benefit of reissue tends to increase for higher target percentiles.

Correlation.

We use the same default Pareto distribution to model service time, and progressively increase the service time correlation ratio r between the primary request and its corresponding reissued request (defined in Section 8.5.1). The $P95$ latency without reissue is 567, and is independent of the correlation ratio r. Figure 8-5a reports $P95$ latency of SingleR using a fixed reissue rate of 25% as a function of the correlation ratio. As expected, the less service-times are correlated the larger benefit reissuing has on tail-latency. Even when primary and reissue requests are strongly correlated (e.g. $r = 1$) SingleR is still able to reduce the response-times of queries delayed due to queueing delays.

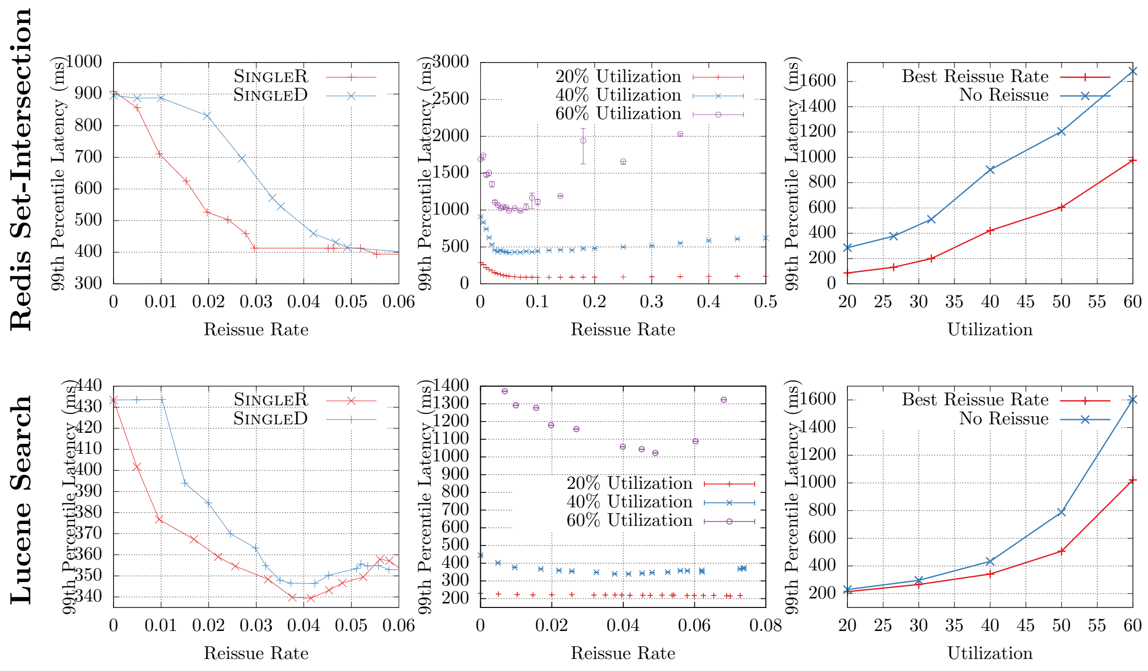
Load-balancing.

Figure 8-5b shows the impact of different load-balancing strategies on tail latency. We make two observations: (1) Using more sophisticated load-balancing strategies such as *Min-of-*2 (select the server with shorter queue among two randomly selected servers to dispatch a request) or *Min-of-All* (select the server with the shortest queue among all servers to dispatch a request) helps reduce the $P95$ tail-latency relative to the simpler *Random* strategy that picks a server uniformly at random. (2) In all cases, SingleR reduces the $P95$th latency by a factor of 2 or more.

Changing priority of reissued requests.

We study three priority settings: (1) *Baseline FIFO* corresponds to a server maintaining a FIFO single queue, and does not differentiate between primary and reissue requests. (2) *Prioritized FIFO* corresponds to a server that maintains two separate FIFO queues for primary and reissue requests, and only processes reissue requests when the primary queue is empty: preventing multiple reissued requests from delaying a primary request. (3) *Prioritized LIFO* is the same as *Prioritized FIFO* but processes the reissue queue in LIFO order. Figure 8-5c compares the three systems and shows that changing the priority scheme has a modest impact on the tail latency improvements of SingleR.

The overall results in this section show that SingleR and its adaptive policy optimizer is robust and reduces the tail latency for these different system design choices and workload characteristics.



(a) SingleR vs SingleD (b) Latency vs Reissue Rate (c) Best Latency vs Utilization

Figure 8-7: System experiment results for the Redis and Lucene workloads. Figure 8-7a compares the $P99$ latency of SingleR and SingleD for reissue budgets between 0 and 6% at 40% utilization. Figure 8-7b shows the $P99$ latency for SingleR with varied reissue rates for 20%, 40%, and 60% utilization. Figure 8-7c shows the $P99$ latency achieved when using the best reissue budget and a SingleR policy for utilizations ranging from 20% to 60%.

8.6 Experimental evaluation

We evaluate SingleR policies in two distributed systems based on Redis [358] which is a key-value store that supports stored procedures, and Lucene [237] which is an enterprise search engine. Our main target is reducing the $P99$ tail latency.

8.6.1 Experimental setup and workloads

We use a cluster of 10 servers to execute the workload. Each server has a dual-core 2.4 GHz Intel E5-2676 processor and 32GB of RAM. The data sets and its associated indices both for Redis and for Lucene fit in the main memory. To execute each query workload, we employ several machines emulating clients that send requests in an open loop with exponential inter-arrival times.

To enable request reissuing, we assign each primary request a timestamp, and add it to a FIFO queue so that the request can be reissued later. A *reissue thread* consumes the entries from the FIFO queue, and dispatches the request to a server after a policy-specified delay. Prior to sending a reissue request, the completion status of its associated query is checked using a client-local boolean array.

All reported system utilizations refer to CPU utilization on a single core as measured by the Linux *sysstat* [130] utility. We use 10 adaptive iterations (with learning rate $\lambda = 0.5$) to compute the SingleD and SingleR policies satisfying the reissue budget. The measured

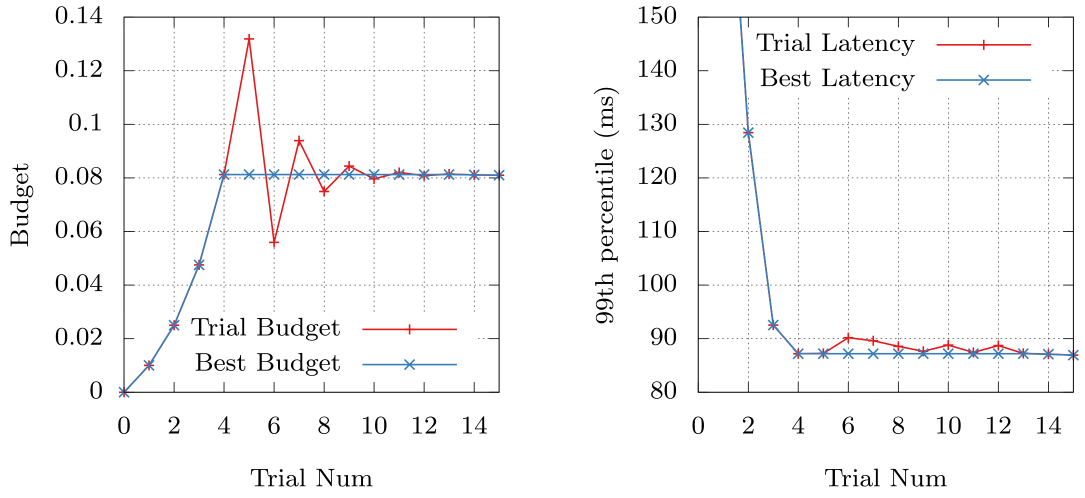


Figure 8-8: Illustration of binary search for optimal budget for the *Intersection Counting* query to minimize 99th percentile tail-latency. **20% utilization**.

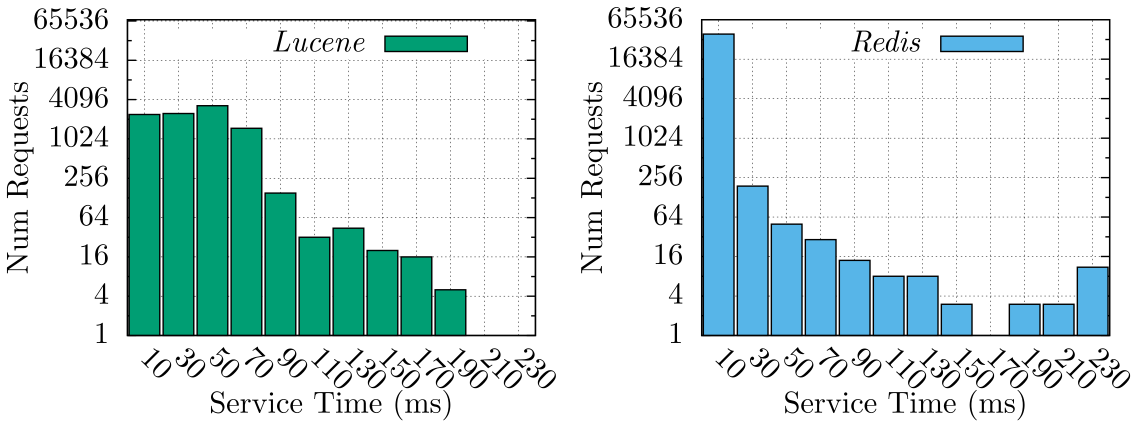


Figure 8-9: Service time distributions for the Redis set-intersection and Lucene search workloads.

reissue rate and the target reissue budget tend to closely agree with the predictions of the reissue policy optimizer and thus we report only the empirical rate in all figures.

8.6.2 Redis set-intersection workload

The Redis workload consists of set-intersection queries performed over a synthetic collection of 1000 sets. Each set stores a random subset of integers in the range 1 and 10^6, and set cardinalities are distributed according to a lognormal distribution. Query traces consist of $40,000$ intersections between randomly selected pairs of sets.

The service-time distribution for the Redis set-intersection workload is illustrated in Figure 8-9 discretized into 20 msec bins. Over 98% of set-intersection queries in this workload have a service-time less than 10 msec. Indeed, the workload's mean service time $\mu_R = 2.366$ milliseconds and standard deviation $\sigma_R = 8.64$ may lead us to expect request latencies to be well-behaved, even in the tail.

A handful of queries (≈ 20), however, have service times greater than 150 msec. These queries correspond to the rare case in which an intersection is performed between two abnormally large sets. These rare queries do little to skew the aggregate statistics of the workload's service-time distribution, but have a substantial impact on tail-latency. As shown in Figure 8-7b, the 99th percentile tail-latency for the set-intersection workload is 900 msec when one does not reissue requests.

These "queries of death" are a common problem in database applications, and their impact on tail-latency can be difficult to predict apriori. In particular, the influence of these requests depends to a large extent on the queueing mechanisms used in the system. In Redis, requests are serviced in a round-robin fashion from each active client connection in a batch. If even a single client issues a long-running request, then the requests of all other clients will be delayed until completion. Furthermore, in an open-loop system such delays lead to a backlog of requests that further extends the impact of the slow request for multiple rounds.

Tail latency reduction under SingleR and SingleD

On the *Redis* workload, the SingleR and SingleD policies are able to reduce the $P99$ latency at 40% utilization from 900 milliseconds to 400 milliseconds. SingleR is able to meet this target $P99$ latency with a budget of just $\approx 3.5\%$, whereas SingleD requires a budget of at least 5%. For reissue budgets between 3 and 5%, the reissue probability of SingleR increases from 0.8 to 1.0 so that for budgets greater than 5% SingleR and SingleD are equivalent.

Figure 8-7a shows the $P99$ latency in msec (Y-axis) against the reissue rates between 0 and 6% (X-axis) for SingleR and SingleD. Both Redis (top figure) and Lucene (bottom figure) running at 40% baseline utilization without any reissue.

We make two observations: First, both the SingleR and SingleD curves illustrate reduced tail latency relative to the baseline system without reissuing. Second, we note that the SingleR policy achieves strictly better tail-latencies than SingleD for small reissue rates. For example at 2% reissue rate in Redis, SingleR reduces the $P99$ latency to 405 msec, compared to 900 msec for the baseline system and 820 msec for SingleD.

Varied system utilization

Next, we illustrate the performance of SingleR under three system utilization levels: 20%, 40%, and 60% in Figure 8-7b, which depicts the $P99$ latency against fixed reissue rate. For all utilizations between $20-60\%$, SingleR is able to reduce the 99th percentile tail latency.

In particular, at 60% utilization (which is very high for interactive services) SingleR with a 3% reissue rate reduces the Redis $P99$ latency from 1750 msec to 1000 msec, and the Lucene $P99$ latency from 1603 msec to 1157 msec.

The best latency reduction occurs when choosing the optimal reissue rate, which depends on the system utilization. For 20% utilization, we illustrate the process of finding the optimal reissue rate via binary search in Figure 8-8. At 20% utilization the best reissue budget is approximately 8%. At both 40% and 60% utilization, the best reissue rate is approximately 5%.

Figure 8-7c illustrates the best tail-latency achieved by a SingleR policy for the Redis workload for utilizations between 20% and 60%. The *Best Reissue Rate* curve illustrates the $P99$ latency achieved by the best SingleR policy (with reissue rate determined via a binary search procedure), and the *No Reissue* curve illustrates the $P99$ latency of the baseline system without reissuing.

8.6.3 Lucene search workload

The Lucene search workload consists of search queries over a corpus of 33 million articles from the English Wikipedia dataset [112]. Queries are drawn randomly from a set of $10,000$ queries from the Lucene nightly regression tests [249].

The service-time distribution for queries in the search workload contrasts with set-intersection in that it is not as highly skewed towards very-low latencies. The distribution for search service-times is illustrated in Figure 8-9 discretized into 20 msec bins. Approximately 90% of all requests have service times between 1 and 70 msec, and the overall distribution has mean service-time $\mu_L = 39.73$ msec with standard deviation $\sigma_L = 21.88$.

Similar to the set-intersection workload, the search workload also has rare slow queries. Approximately 1% of search queries have service-times greater than 100 msec. The impact of these slow queries on tail-latency, however, is different in Lucene than in Redis. At 40% utilization, the search workload's 99th percentile latency is ≈ 435 msec when there are no reissued requests. This is not entirely due to the differences in service-time distribution, although it certainly is an important influence. The Lucene search server also differs in how it manages concurrent requests. Requests from all open connections are placed into a single FIFO queue which results in relatively good tail-latency behavior — FIFO is, in fact, optimal for light-tailed service-time distributions [344].

Tail-latency reduction under SingleR and SingleD

On the *Lucene* workload, we observe in Figure 8-7a that SingleR reduces Lucene's $P99$ latency at 40% utilization from 433 milliseconds to 339 milliseconds, and the SingleD policy reduces $P99$ to 346 milliseconds. This gap, while small, is not merely measurement noise — all reported values reflect the median of multiple runs.

The improved performance of the SingleR policy is due to its use of randomization that allows it to reissue queries earlier than SingleD. At 40% utilization, the optimal reissue rate for SingleR is 4%, and the optimal policy reissues requests with probability approximately 0.75. As the reissue rate grows the achieved latency gap between SingleR and SingleD closes, and the reissue probability of the optimal SingleR policy converges to 1.0. Randomization is more valuable on the Lucene search workload than it was for Redis because of the much higher mean service time of requests.

Varied system utilization

Next, we illustrate the performance of SingleR under three system utilization levels: 20%, 40%, and 60% in Figure 8-7b, which depicts the $P99$ latency against fixed reissue rate.

SingleR reduces the tail-latency of Lucene search workload for all utilizations between $20 - 60\%$. At 60% utilization (high load), SingleR reduces the $P99$ latency from 1603 to 1157 msec. Figure 8-7c illustrates the best tail-latency achieved by a SingleR policy for the Lucene workload for utilizations between 20% and 60%. The *Best Reissue Rate* curve illustrates the $P99$ latency achieved by the best SingleR policy (with reissue rate determined via a binary search procedure), and the *No Reissue* curve illustrates the $P99$ latency of the baseline system without reissuing. We observe significant tail latency reduction due to SingleR over the baseline.

8.7 Conclusion

We have illustrated principled methods of generating reissue policies for interactive services. By operating within a simplified model, we were able to prove that SingleR is an optimal compromise between the commonly used immediate and delayed reissue strategies. There are a few general lessons that we think are useful to impart: a) there is little reason to choose a reissue policy more complex than SingleR if that additional complexity does not leverage application-specific insight; and, b) reissue policies that reduce tail-latency as a "smooth" function of their budget admit relatively simple strategies for adapting to load-dependent queueing delays and searching for optimal reissue budgets. As we have shown, we were able to adapt SingleR policies to systems and workloads with a wide range of properties through iterative adaptation. As we have seen, this leads to a simple process for finding effective reissue policies in real systems: SingleR is able to reduce tail-latency in simulated and real-world workloads even when reissuing a small fraction of requests.

Chapter 9

Polylogarithmic Fully Retroactive Priority Queues via Hierarchical Checkpointing

This chapter presents a fully-retroactive priority-queue data struture that supports updates and queries on a timeline containing m updates with overheads that are polylogarithmic functions of m. This improves upon the previous best-known fully retroactive priority queue data structure that relied upon a general transformation that required $\Theta(\sqrt{m}\log m)$ time per operation. This work was conducted in collaboration with Erik D. Demaine, Quanquan Liu, Aaron Sidford, and Adam Yedidia.

Abstract

Since the introduction of retroactive data structures at SODA 2004 [88], a major open question has been the difference between partial retroactivity (where updates can be made in the past) and full retroactivity (where queries can also be made in the past). In particular, for priority queues, partial retroactivity is possible in $O(\log m)$ time per operation on a m-operation timeline, but the best previously known fully retroactive priority queue has cost $\Theta(\sqrt{m}\log m)$ time per operation.

We address this open problem by providing a general logarithmic-overhead transformation from partial to full retroactivity called "hierarchical checkpointing," provided that the given data structure is "time-fusible" (multiple structures with disjoint timespans can be fused into a timeline supporting queries of the present). As an application, we construct a fully retroactive priority queue which can insert an element, delete the minimum element, and find the minimum element, at any point in time, in $O(\log^2 m)$ amortized time per update and $O(\log^2 m \log\log m)$ time per query, using $O(m\log m)$ space. Our data structure also supports the operation of determining the time at which an element was deleted in $O(\log^2 m)$ time.

9.1 Introduction

Retroactivity. We can think of a data structure as being defined by a sequence of updates $u_1, u_2, \ldots, u_m$ applied to its initial (empty) state. Traditional data structures "live in the present" in the sense that the user can only append updates to this sequence, and ask queries about the final state of the data structure resulting from the entire update sequence. ***Retroactive data structures***, introduced at SODA 2004 [88], allow for updates to be inserted or deleted in the middle of the sequence, instead of just the end. Effectively, this feature enables the user to travel back in time and make a retroactive change to the data structure (similar to the movie *Back to the Future*). Thus we refer to the mutable update sequence as the ***timeline***.

We distinguish two forms of retroactivity. In ***partial retroactivity***, queries can be made only of the final version resulting from all of the updates in the timeline; effectively, retroactive updates must be propagated all the way through the timeline in order to answer such queries correctly. In ***full retroactivity***, queries can be made about the data structure at any time, corresponding to the result from a prefix of the timeline. In short, both forms of retroactivity enable modifying the past, and full retroactivity enables querying the past.

Known results. In some settings, retroactivity is easy to achieve. If updates commute with each other and have inverses, then retroactive updates can be moved to the end of the timeline, making partial (but not full) retroactivity easy. If updates are inserts and deletes, and the queries fall under Bentley and Saxe's decomposable search problems, then full retroactivity is possible with an $O(\log m)$ factor overhead [88].

Retroactivity becomes challenging when updates can have non-trival interactions. Here one retroactive update can have a propagated effect on potentially all later updates. In the extreme, when the data structure is a general-purpose computer, a retroactive update can require an $\Omega(m)$ factor overhead [88].

The more interesting middle ground is when the updates have some but limited influence on each other—a common scenario in many classic data structures. For example, logarithmic fully retroactive stacks (with push/pop), queues (with enqueue/dequeue), deques (with all four), union-find, dictionaries, and predecessor/successor structures all have logarithmic fully retroactive data structures [88, 127]. Of these results, predecessor/successor was the most challenging; the original paper [88] solved partial retroactivity in $O(\log m)$ but full retroactivity in $O(\log^2 m)$, which was later improved to $O(\log m)$ by Giora and Kaplan [127]. This problem is equivalent to dynamic rectilinear ray shooting, which was in fact the original motivation for defining retroactivity.

Challenges. A key open problem in retroactivity, posed at SODA 2004, is whether there is a difference in difficulty between obtaining partial versus full retroactivity. The only known upper bound on the separation is a conversion from partial to full retroactivity with $O(\sqrt{m})$ factor overhead [88]. Essentially, this conversion maintains $\Theta(\sqrt{m})$ checkpoints of the timeline using a partially retroactive data structure, and to query in between, replays the necessary $O(\sqrt{m})$ intervening updates. On the other hand, the only known data structural problem with a polynomial separation between the best partially retroactive and best fully retroactive data structures is priority queues (with insert and delete-min operations). The logarithmic partially retroactive priority queue [88] is one of the most sophisticated retroactive data structures, propagating potentially linear-length chain reactions in just logarithmic time. However, the existing approach appeared limited to partial retroactivity. Until now, the fastest known fully retroactive priority queue was the $O(\sqrt{m} \log m)$ bound that follows from the general conversion.

Our results. In this chapter, we solve this 11-year-old open problem by constructing the first polylogarithmic fully retroactive priority queue. Specifically, our data structure supports inserting an element, deleting the minimum element, and finding the minimum element, at any time in the timeline, in $O(\log^2 m)$ amortized time per update and $O(\log^2 m \log\log m)$ time per query, using $O(m \log m)$ space. We also show how to support another natural query over the timeline: finding the time at which a given element gets deleted as the minimum (or finding that it remains in the structure in the present).

More importantly, we present a new general transformation from partial to full retroactivity with only a logarithmic factor overhead. This result shows a strong upper bound on the separation between partial versus full retroactivity, but it requires one additional assumption. Specifically, we call a (partially retroactive) data structure ***time-fusible*** if, given two such data structures representing two different timelines (contained in disjoint time intervals), it is possible to form a new (read-only) data structure representing the concatenation of those timelines. Roughly speaking, this assumption lets us apply the $O(\sqrt{m})$ checkpointing idea recursively in a binary tree structure built over the timeline, storing a partially retroactive data structure for the sub-timeline represented by each rooted subtree. Hence we call the transformation ***hierarchical checkpointing***. A retroactive query can then be answered by fusing $O(\log m)$ structures and asking a query about the present.

Our fully retroactive priority queue data structure is an application of this general technique. With some modifications, we show how to fuse two of the logarithmic partially retroactive priority queues from [88] in polylogarithmic time. Applying the general technique gives us a polylogarithmic bound on fully retroactive priority queues, but with worse bounds than those stated above. By a more careful analysis tailored to priority queues, we show how to further tune the hierarchical checkpointing analysis to improve the running time by a logarithmic factor and get the claimed bounds of $\tilde{O}(\log^2 m)$.

Organization. We organize the sections of this chapter as follows. Section 9.2 introduces our hierarchical checkpointing framework in greater detail. Section 9.3 describes time-fusible partially retroactive priority queues whose timelines may be fused together in polylogarithmic time. Section 9.4 applies the technique of hierarchical checkpointing to obtain a fully retroactive priority queue with polylogarthmic overheads.

9.2 Hierarchical checkpointing

In this section, we present our hierarchical checkpointing technique for transforming a time-fusible partially retroactive data structure into one that is fully retroactive while incurring only polylogarithmic overheads. In later sections, these results will be employed to design a fully retroactive priority queue with polylogarithmic overheads.

We begin by defining in Section 9.2.1 the notion of time fusibility for retroactive data structures. Then in Section 9.2.2 we describe the hierarchical checkpoint procedure and prove its correctness.

9.2.1 Definitions

Here we discuss the properties of partially retroactive data structures and the conditions necessary to use hierarchical checkpointing to obtain full retroactivity.

We define a ***retroactive update*** operation to be the insertion or deletion of a data structure operation at a particular time. These operations are:

- Insert-Op(o, t): insert a data structure update operation o into the retroactive structure's timeline at time t.
- Delete-Op(o, t): delete a data structure update operation o from the retroactive structure's timeline at time t.

We define a ***retroactive query*** operation to be one that can determine some aspect of the state of the retroactive data structure at some point in time. We use Get-View(t) as the canonical query procedure when we describe our transformation.

- Get-View(t): returns some aspect of the state of the retroactive data structure at time t.

For partially retroactive structures, query operations can only be performed in the present (i.e. $t = \infty$). Fully retroactive data structures, however, may be queried at any time t. It turns out, that a collection of partially retroactive data structures can be used to support fully retroactive query operations when it is possible to "fuse" their timelines. Formally, we say a partially retroactive data structure is ***time fusible*** if it has the following properties:

1. It supports a function, Fuse(d_1, d_2), that fuses the timelines of two instances d_1 and d_2 of the partially retroactive data structure, producing a version of the data structure that allows read-only queries and reflects the updates in both d_1 and d_2. Fuse(d_1, d_2) need only support fusion between structures containing updates that span disjoint and adjacent intervals of the timeline.
2. Sequences of operations made on it exhibit substring closure; in other words, given a valid sequence of operations, any contiguous subsequence of operations on the structure is also valid.

9.2.2 The data structure

In this section we describe how to transform a time-fusible partially retroactive data structure into one that is fully retroactive using our hierarchical checkpointing framework. Specifically, we obtain a fully retroactive data structure with $O(T(m)\log m + Q(m, k))$ query time and $O(A(m)\log^2 m)$ amortized update time, where $T(m)$ and $A(m)$ represent the merge and update time, respectively, in the original partially retroactive data structure, and $Q(m, k)$ is the query time of a time-fused structure consisting of k fusions and containing m updates.

The first step of our transformation is to build a ***checkpoint tree*** — a balanced binary search tree in which each node of the tree contains a partially retroactive data structure consisting of all the updates in the subtree rooted at that node. Our checkpoint tree is similar to a segment tree [21] in that each partially retroactive data structure can be viewed as a segment with endpoints given by the first and last chronological update in the structure. The structures in the leaves of our checkpoint tree each contain only one update, and the leaves are sorted by the time of their one update. The update operations Insert-Op(o, t) or Delete-Op(o, t) can be performed on the fully retroactive structure by inserting into or deleting the update, o, from all of the partially retroactive structures in the search path. A query can be performed at time t by merging $O(\log n)$ disjoint partially retroactive structures obtained from the balanced binary tree such that the fused structure contains all updates in the time span $(-\infty, t]$.

Theorem 47 *Given a partially retroactive data structure that is time fusible, we may construct a fully retroactive version of the data structure using hierarchical checkpointing. This data structure will have an $O(A(m)\log^2 m)$ amortized update time and $O(T(m)\log m + Q(m,k))$ query time.*

We prove Theorem 47 in two parts below.

Lemma 48 *Our hierarchical checkpointing method produces a fully retroactive data structure with $O(A(m)\log^2 m)$ amortized update time.*

Proof. Let F be a fully retroactive data structure based on a time-fusible partially retroactive data structure P. Suppose that m updates have been inserted into F and that the update operation for P runs in $O(A(m))$ time.

We utilize a scapegoat tree [117] to represent the checkpoint tree for F. The checkpoint tree contains all updates to the fully retroactive structure at its leaves ordered by time. Each internal node, x, is associated with an instance of P that reflects the application of all updates in its subtree. To perform Insert-Op(o, t) or Delete-Op(o, t), we insert the update as a leaf in the checkpoint tree, and apply the update to the instances of P associated with nodes along the roof to leaf path in $O(A(m)\log m)$ time.

To rebalance the checkpoint tree, the tree rooted at the scapegoat node is rebuilt. We begin by obtaining a sorted list of the k updates ordered by time by performing an in-order walk of the subtree. We create a balanced binary tree with these k updates at the leaves, and initialize an empty instance of P for each internal node of the subtree. Then, we insert the update at each leaf into each of its $O(\log k)$ ancestors. Because applying an update to an instance of P takes $O(A(k))$ time, the total time required to rebuild a subtree containing k updates is $O(A(k)\log k)$. The total cost of an Insert-Op or Delete-Op operation for the fully retroactive structures is then the sum of the cost of an insertion or deletion and the amortized cost of rebuilding, $O(A(m)\log^2 m)$ amortized.

□

Lemma 49 *Our hierarchical checkpointing method produces a fully retroactive data structure with $O(T(m)\log m + Q(m,k))$ query time.*

Proof. Suppose that $T(m)$ is the time it takes to fuse any two instances of P, and $Q(m,k)$ is the time it takes to query an instance of P, where m is the total number of updates in P, and k is the number of components that were used to create the fused structure.

To perform Get-View(t), we first traverse the checkpoint tree to identify the $O(\log m)$ disjoint subtrees that represent the time interval $(-\infty, t]$. The time-fusible partially retroactive structures associated with these subtrees are then fused in-order, resulting in a single structure representing the interval $(-\infty, t]$. We can fuse $O(\log m)$ P structures in $O(T(m)\log m)$ time. Querying this structure then takes $O(Q(m,k))$ time. Therefore, the total runtime of Get-View(t) is $O(T\log m + Q(m,k))$.

□

9.3 Time-fusible partially retroactive priority queue

In this section we present a partially retroactive priority queue that supports a polylogarithmic fusion operation. Specifically, we describe an algorithm that fuses $k = O(\log m)$ partially retroactive priority queues containing m updates in $O(k \log k \log m)$ time. This time-fusible partially retroactive priority queue enables the use of hierarchical checkpointing to obtain a fully retroactive priority queue with polylogarithmic overheads.

9.3.1 Partially retroactive priority queues

We begin with an informal review of a partially retroactive priority queue data structure. To simplify our exposition, we treat the partially retroactive priority queue from [88] as a black box and maintain 2 auxillary data structures: Q_{now} containing the set of all keys remaining in the priority queue at time $t = \infty$, and Q_{del} containing all keys that were removed from the priority queue at some point in the past.

We assume that the partially retroactive priority queue returns, following each retroactive update, the keys which should be inserted or deleted from Q_{now} and Q_{del}. If a priority queue is empty at time t, then a delete-min operation will, by convention, insert a placeholder key of infinite weight into Q_{del}. It is known that, following a retroactive update at time t, it is only necessary to insert or delete a single key into Q_{now} and Q_{del} [88]. We can, therefore, synchronize our auxillary data structures Q_{now} and Q_{del} with the partially retroactive priority queue in $O(\log m)$ time. A proof of this claim and an in-depth description of the partially retroactive priority queue data structure can be found in [88, 5.4].

The auxillary Q_{now} and Q_{del} structures are maintained using weight-balanced B-trees [323, 19, 9] which for a balance factor $d > 4$ have the following properties:

- Insertion and deletion operations on a B-tree containing m elements take $O(\log m)$ time.
- For all non-root nodes u at height h the weight $w(u)$ of the subtree rooted at u is bounded as follows: $d^h/2 \leq w(u) \leq 2d^h$.
- The root r of a height-h tree has bounded weight $w(r)$: $d^{h-1} \leq w(r) \leq 2d^h$.
- Tree-split and concatenate operations on a size-m tree take $O(\log m)$ time.
- A height-h' subtree T' of a height-h weight-balanced B-tree T can be deleted to form the weight-balanced B-tree $T - T'$ in $O(d(h - h'))$ time.

A weight-balanced B-tree data structure possessing these properties is described in [19, 9]. Specifically, we apply the result of [19] with balance factor $d = 8$ to maintain Q_{now} and Q_{del}.

9.3.2 Fusion algorithm

Before describing our algorithm for fusion, let us better understand the structure of the problem by proving a mathematical relationship between two partially retroactive priority queues that represent two fusible (i.e. disjoint and adjacent) intervals of time.

Lemma 50 *Consider two partially retroactive priority queues Q_1 and Q_2 whose update times lie in the intervals $[a, b)$ and $[b, c)$ respectively. Then, the partially retroactive priority queue Q_3 containing all updates in Q_1 and Q_2 in the interval $[a, c)$ has the property that*

$$Q_{3,now} = Q_{2,now} \cup \textit{max-}A\{Q_{1,now} \cup Q_{2,del}\} \tag{9.1}$$

$$Q_{3,del} = Q_{1,del} \cup \textit{min-}D\{Q_{1,now} \cup Q_{2,del}\} \tag{9.2}$$

where $A = |Q_{1,now}| - |Q_{2,del}|$, $D = |Q_{2,del}|$ and max-$C\{S\}$ denotes the C largest elements in the set S.

```
GetSplitKey($s, T_1, \ldots, T_k$)
  1. If $N = \sum_i |T_i| < C$ (for constant $C$), sort $\bigcup_i T_i$ and
     return the $s$th element.
  2. If $s < N/2$, set $s = N - s$ and "invert" the order of
     each $T_i$.
  3. For each $T_i$, pick a leftmost subtree $T_{m_i}$ containing
     keys in the range $(-\infty, m_i)$ where $m_i$ has
     an order statistic in $T_i$ contained in the range
     $(|T_i|/256, |T_i|/4)$.
  4. Assign each $m_i$ the weight $w_i = |T_i|$. Using
     weighted selection, select the $N/4$th element $m_j$
     among $m_1, m_2, \ldots, m_k$
  5. For $m_i \leq m_j$, let $T'_i = T_i - T_{m_i}$. For $m_i > m_j$, let
     $T'_i = T_i$.
  6. Set $s_{new} = s - \sum_i(|T_i| - |T'_i|)$ and return
     GetSplitKey($s_{new}, T'_1, \ldots, T'_k$).
```

(a) Psuedocode for the GetSplitKey operation.

```
Fuse($Q_1^k, Q_2^k$)
  1. $A = |Q_{1,now}| - |Q_{2,del}|$
  2. Form a list of $2k$ trees $L = T_1, \ldots, T_{2k}$ by
     concatenating the list of $k$ trees representing
     $Q_{1,now}$ with the $k$ trees representing
     $Q_{2,del}$.
  3. $x$ = GetSplitKey($A, T_1, \ldots, T_{2k}$)
  4. For $i = 1, 2, \ldots 2k$, split the tree $T_i$ on the
     key $x$ to obtain 2 trees $T_{i,>}$ and $T_{i,<}$.
  5. $Q_{3,now} = Q_{2,now} + T_{1,>}, \ldots, T_{2k,>}$
  6. $Q_{3,del} = Q_{1,del} + T_{1,<}, \ldots, T_{2k,<}$
  7. Return $Q_3$
```

(b) Psuedocode for the Fuse operation.

Figure 9-1: Pseudocode for (a) the GetSplitKey operation; and (b) the Fuse operation. GetSplitKey takes a key s and a list of k binary trees, and returns a key x such that s keys in $T_1, T_2, \ldots, T_k$ are less than x.

Proof. Note that fusing Q_1 with Q_2 is equivalent to retroactively inserting all keys in $Q_{1,now}$ into the timeline of Q_2 at $t = -\infty$. With this observation, the lemma can be shown to follow from an iterative application of [88, Lemma 6]. Here we present an intuitive sketch of the proof.

First observe that all keys in $x \in Q_{1,del}$ are also deleted in the fusion Q_3. Next, suppose there is a key $x \in Q_{1,now}$ that is smaller than the maximum element d_{max} in $Q_{2,del}$. If we insert x into Q_2 at time $t = -\infty$, then any delete-min in Q_2's timeline will strike x before it strikes d_{max}. The element d_{max} is never deleted after the insertion of x, because all keys d' deleted after d_{max} are less than d_{max}. We conclude that $Q_{3,del}$ contains the elements in $Q_{1,del}$ and the minimum $D = |Q_{2,del}|$ elements of $Q_{1,now} \cup Q_{2,del}$.

To determine the contents of $Q_{3,now}$ we observe that no element in $Q_{2,now}$ can be deleted as the result of adding additional elements at $t = -\infty$, and observe that keys which were inserted, but never deleted from Q_3's timeline must be in $Q_{3,now}$. It follows that $Q_{3,now}$ contains the contents of $Q_{2,now}$ and the maximum $A = |Q_{1,now}| - |Q_{2,del}|$ elements of $Q_{1,now} \cup Q_{2,del}$. □

Using Lemma 50 we can construct a time-fused representation of Q_3 from Q_1 and Q_2 in polylogarithmic time. We will represent each of $Q_{3,now}$ and $Q_{3,del}$ as a list of trees obtained via tree-split operations consistent with the application of Equation (9.1) and Equation (9.2). We say that a time-fusible partially retroactive priority queue has order k, and use the superscript notation Q^k, if Q^k_{now} and Q^k_{del} are represented as lists of at most k trees.

In Figure 9-1 we provide the pseudocode for Fuse which fuses two partially retroactive priority queues Q_1^k, Q_2^k to obtain Q_3^{3k}. Step 1 computes the value of A from Lemma 50, and Step 2 concatenates the list of trees representing $Q^k_{1,now}$ and and $Q^k_{2,del}$ to form a list L containing $2k$ trees. Step 3 computes a "split-key" x that is greater than A elements contained in trees of L. Next each tree in L is split in Step 4 by performing a tree-split operation to divide each tree T_i into a tree $T_{i,<}$ containing all keys in T_i that are less than x and $T_{i,>}$ containing all keys in T_i that are greater than x. The trees $T_{i,>}$

for $i = 1, 2, \ldots, 2k$ combined with the trees in $Q_{2,now}$ contain the elements satisfying the relation of Equation (9.1) in Lemma 50, and similarly the trees in $Q_{1,del}$ and in $T_{i,<}$ for $i = 1, 2, \ldots, 2k$ contain the elements satisfying the relation of Equation (9.2).

The following theorem proves that FUSE fuses two partially retroactive priority queues of order k in $O(k \log m)$ time.

Theorem 51 *Consider two partially retroactive priority queues Q_1^k and Q_2^k with order k containing m operations. Then* FUSE(Q_1^k, Q_2^k) *runs in $O(k \log m)$ time.*

PROOF.

We first show that GETSPLITKEY runs in $O(k \log m)$ time. Our algorithm for finding the split key is an adaptation of the approach of Frederickson and Johnson to compute order statistics for sorted arrays [113].

Steps 1, 2, and 4 of GETSPLITKEY run in $O(k)$ time (step 4 uses linear-time weighted selection from [290]).

Step 3 finds a leftmost subtree T_{m_i} whose contents are contained in the range $(-\infty, m_i)$ and where the order statistic of m_i is in the range $(|T_i|/256, |T_i|/4)$. We show that step 3 runs in $O(k)$ time by showing that for each T_i such a subtree exists at a distance of at most 2 from the root. Consider a height-h weight-balanced B-tree with balance factor d, root node r, and an internal node u at height $h-2$. The weight-balance criteria for B-trees provided in Section 9.3.1 implies that the ratio $w(u)/w(r)$ is bounded in the range $(1/256, 1/4)$. The key m_i can, therefore, be found in $O(1)$ time by selecting the maximum key from the leftmost height-$(h-2)$ subtree of T_i.

Step 5 deletes the subtree T_{m_i} from T_i if $m_i \leq m_j$. The difference in the heights of T_{m_i} and T_i is at most 2, which allows $T - T_{m_i}$ to be obtained in $O(d)$ time while preserving weight-balance. For $d = 8$, this step runs in $O(k)$ time. Note that the subtrees deleted in this step contain elements whose order statistic is strictly less than $N/2$ and thus these subtrees can not contain the sth order statistic. To prove this we show that the order statistic of m_j, computed in step 4, is less than $N/2$. The key m_j is selected in step 4 such that $3N/4$ elements are contained in trees T_i for which $m_i > m_j$. For each such i, the key m_i is smaller than at least $3|T_i|/4$ of the elements in T_i. The key m_j is, therefore, smaller than at least $9N/16$ elements, and thus has an order statistic less than $N/2$.

Step 6 updates the value of s to reflect the reduced problem size, and recursively calls GETSPLITKEY. To bound the depth of the recursion, it is sufficient to show that step 5 eliminates a constant fraction of the elements. Since a total of $N/4$ elements are contained in trees T_i for which $m_i \leq m_j$, and at least $|T_i|/256$ elements in T_i are smaller than m_i, step 5 eliminates at least $N/1024$ elements. The recursion depth is, therefore, bounded by $O(\log N)$. Since $N = O(m)$, the total runtime of GETSPLITKEY is $O(k \log m)$

Next let us analyze the FUSE operation. Steps 1-2 and 5-6 of FUSE can be performed in $O(k)$ time. Step 3 to compute the split key runs in $O(k \log m)$ time, and step 4 may be performed in $O(k \log m)$ time by performing an $O(\log m)$ time tree split operation on each of k trees. The runtime of FUSE is bounded by the time to compute the split key, and therefore is $O(k \log m)$.

□

The bound proved in Theorem 51 depends on the order k of the two time-fusible partially retroactive priority queues Q_1^k, Q_2^k being merged. It turns out, that the fusion of k partially retroactive priority queues can be constructed efficiently while being represented using only

$O(k)$ trees by combining trees in Q_{now} and Q_{del} that originated from a split operation on a common tree. The ability to perform such a reduction relies on the following lemma.

Lemma 52 *Let $Q_1, \ldots, Q_k$ denote k partially retroactive priority queues each with disjoint time intervals that increase monotonically with k. Let Q_* be a priority queue containing the updates in $Q_1, \ldots, Q_k$ applied consecutively. Then $Q_{*,now}$ and $Q_{*,del}$ consist of contiguous intervals of $Q_{i,now}$ and $Q_{i,del}$, i.e.*

$$Q_{*,now} = \cup_{i \in S_{now}} Q_{i,now}[a_i, b_i] \cup_{i \in S_{del}} Q_{i,del}[a'_i, b'_i] \tag{9.3}$$

$$Q_{*,del} = \cup_{i \in T_{now}} Q_{i,now}[c_i, d_i] \cup_{i \in T_{del}} Q_{i,del}[c'_i, d'_i] \tag{9.4}$$

for some sets $S_{now}, S_{del}, T_{now}, T_{del} \subseteq \{1, \ldots, k\}$ and elements a_i,a'_i,b_i,b'_i,c_i,c'_i,d_i,d'_i where for a set S and $a, b \in S$ we let $S[a,b] = \{x \in S : a \leq x \leq b\}$.

Proof. The proof follows almost immediately from Lemma 50. We prove it by induction on k. Note that when $k = 1$ the statement is trivial as clearly in this case $Q_{*,now} = Q_{1,now}$ and $Q_{*,del} = Q_{1,del}$.

Now suppose we wish to prove the lemma for the given $k > 1$ and we know that it holds for $k-1$. Let Q' denote the priority queue that contains the updates in $Q_1, \ldots, Q_{k-1}$ applied consecutively. By the inductive hypothesis we know that Q'_{now} and Q'_{del} consist of contiguous intervals of $Q_{i,now}$ and $Q_{i,del}$ for $i \leq k$. Applying Lemma 50 we have that

$$Q_{*,now} = Q_{k,now} \cup \text{max-}A\{Q'_{now} \cup Q_{k,del}\} \quad \text{for} \quad A = |Q'_{now}| - |Q_{k,del}| \tag{9.5}$$

and

$$Q_{*,del} = Q'_{del} \cup \text{min-}B\{Q'_{now} \cup Q_{k,del}\} \quad \text{for} \quad B = |Q_{k,del}| \tag{9.6}$$

However, as $[a,b] \cap (-\infty, c] = [a, \min\{b,c\}]$ and $[a,b] \cap [c, \infty) = [\max\{a,c\}, b]$ we see that $Q_{*,now}$ and $Q_{*,del}$ each consist of contiguous intervals of the $Q_{i,now}$ and $Q_{i,del}$ (here the intervals are the intervals within the Q'_{now} as well as $Q_{k,del}$ and the inequality constraint is the one imposed by restricting to max-A or min-B). Note that this argument uses that Equation (9.5) and Equation (9.6) each consist of the union disjoint sets and therefore disjoint intervals.

Thus we have shown that the result holds for $k = 1$ and we have shown that if the result holds for $k-1$ then it holds for k. The result follows by induction. □

The preceding lemma allows us to tweak the fusion algorithm to guarantee that the order of the fusion of k time-fusible partially retroactive priority queues is bounded by $2k$. This is accomplished by adding a post-processing step PostFuse immediately after the fusion procedure Fuse. After obtaining the fusion Q_3 of Q_1 and Q_2, the trees representing $Q_{3,now}$ are checked in PostFuse to identify pairs of split-trees that were obtained by splitting a common tree. By Lemma 52 the union of these intervals span disjoint intervals and these pairs of trees can, therefore, be concatenated in logarithmic time.

Lemma 53 *The fusion of k time-fusible partially retroactive priority queues has order bounded by $2k$ and runs in $O(k \log m)$ time when using the* PostFuse *procedure.*

Proof. We utilize the Fuse procedure to fuse 2 time-fusible partially retroactive priority queues Q_1^k and Q_2^k to obtain Q_3^{2k}.

Following the fusion, the PostFuse procedure identifies pairs of trees in $Q_{3,now}^{2k}$ which originated from a previous split operation on a common tree T. These pairs contain keys that span disjoint intervals I_1, I_2, and thus can be concatenated to form a single tree in $O(\log m)$ time. This is done by sorting such trees are sorted in $O(k \lg k)$ time by their associated interval, and then performing concatenation in $(\log m)$ time.

By Lemma 52 the keys in the concatenated tree form a contiguous interval $I_1 \cup I_2$ in T and, as such, performing the concatenation does not limit the algorithm's ability to perform additional concatenations between intervals in T.

The time required to perform ProcFuse is dominated by the time to perform Fuse, and therefore the time bound follows from Theorem 51. □

To combine the results of this section, we prove the following theorem.

Theorem 54 *Consider $k = O(\log m)$ time-fusible partially retroactive priority queues. The time to fuse these k data structures is bounded by $O(k \log k \log m)$, and the time required to query this structure is $O(\log^2 m)$.*

Proof. We arrange the k time-fusible structures at the leaves of a balanced height-$\log k$ merge tree. By Lemma 53 the sum of the orders of time-fusible partially retroactive priority queues at level i in the merge tree is $O(k)$. The total work to perform fusions at level i is, therefore, $O(k \log m)$ Since there are $\log \log m$ levels in the merge tree the total time is $O(k \log m \log \log m)$. To query the fused structure we perform a query on each of the $O(\log m)$ trees representing Q_{now} which can be done in $O(\log^2 m)$ time. □

9.4 Fully retroactive priority queue

In this section we describe the design of a fully retroactive priority queue that uses hierarchical checkpointing. We begin in Section 9.4.1 by showing how to apply our technique of hierachical checkpointing using the time-fusible partially retroactive priority queue of Section 9.3. This yields a fully retroactive priority queue that supports retroactive updates in $O(\log^3 m)$ amortized time, retroactive queries in $O(\log^2 m \log \log m)$ time, and Find-Deletion-Time in $O(\log^3 m \log \log m)$ time. Next, in Section 9.4.2, we optimize our application of hierarchical checkpointing for priority queues to obtain $O(\log^2 m)$ amortized time updates, and $O(\log^2 m)$ time Find-Deletion-Time queries.

9.4.1 Obtaining full retroactivity using hierarchical checkpointing

Here we analyze the fully retroactive priority queue obtained by a straightforward application of hierarchical checkpointing. The time-fusible partially retroactive priority queue described in Section 9.3 meets the prerequisites of Theorem 47 needed to perform the partial-to-full transformation. Consequently we can directly apply this theorem to obtain a fully retroactive priority queue which follows the structure laid out in Section 9.2. A checkpoint tree contains all retroactive updates ordered by time, and each internal node maintains a time-fusible partially retroactive priority queue that contains the updates within its subtree.

The checkpoint-tree data structure used in this fully retroactive priority queue is shown in Figure 9-2(a) after 16 retroactive operations have been performed. In this example, the checkpoint tree has 16 leaves each corresponding to a retroactive operation on the priority queue. The time-fusible partially retroactive priority queue data structure described in Section 9.3 is used to represent the partial checkpoints in a checkpoint tree. Each internal node, $Q_{[a,b)}$, maintains a time-fusible partially retroactive priority queue that contains all retroactive operations in its subtree (i.e. all operations occurring at times $t \in [a, b)$).

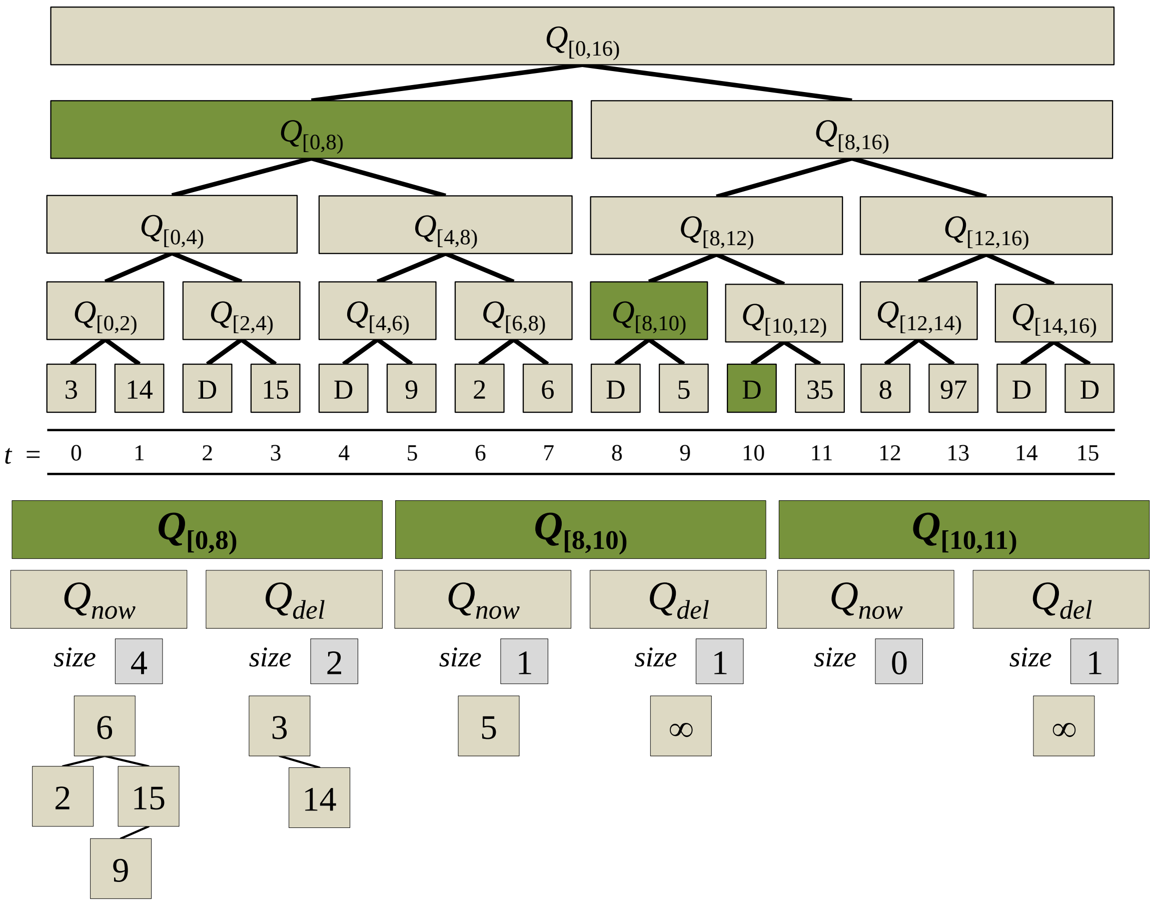


Figure 9-2: Hierarchical checkpointing for fully retroactive priority queue. Illustration of the checkpoint tree for a fully retroactive priority queue with 16 operations.

The GET-VIEW(t) operation is illustrated in Figure 9-2(b). A checkpoint representing the priority queue at time $t = 10$ is constructed by combining 3 partial checkpoints from the checkpoint tree. The time-fusible partially retroactive priority queues $Q_{[0,8)}$, $Q_{[8,10)}$, and $Q_{[10,11)}$ that are highlighted in Figure 9-2 are collected and then merged to obtain obtain a partially retroactive priority queue containing all updates in in the interval $[-\infty, 10]$.

Theorem 55 *There exists a fully retroactive priority queue that supports retroactive updates in $O(\log^3 m)$ amortized time, queries in $O(\log^2 m \log\log m)$, and the operation,* FIND-DELETION-TIME*, in $O(\log^3 m \log\log m)$ time.*

PROOF.

The time-fusible partially retroactive priority queue described in Section 9.3 supports retroactive updates in $O(\log m)$ time. Applying Lemma 48 with $A(m) = \log m$ shows that retroactive updates run in $O(\log^3 m)$ amortized time. By Theorem 54, the time to merge $O(\log m)$ time-fusible partially retroactive priority queues is bounded by $O(\log^2 m \log\log m)$. Similarly, the time to query this merged structure is bounded by $O(\log^2 m)$ since the merged priority queue has order $O(\log m)$. Applying Lemma 49 with $T(m) = O(\log^2 m)$ and $Q(m) = O(\log^2 m \log\log m)$ shows that retroactive queries run in $O(\log^2 m \log\log m)$ time. Finally, the FIND-DELETION-TIME(x) operation can be performed via binary search to identify the first time t for which the key x is not in the queue. This involves $O(\log m)$

retroactive queries showing that FIND-DELETION-TIME runs in $O(\log^3 m \log\log m)$ time.
□

9.4.2 Faster retroactive updates and FIND-DELETION-TIME queries

The general transformation described in Section 9.2 maintains balance in the checkpoint tree by reapplying all updates in rebuilt subtrees. As shown in Lemma 56 a checkpoint tree for priority queues can be rebuilt more efficiently.

Lemma 56 *A subtree of the fully retroactive priority queue's checkpoint tree containing m updates can be rebuilt in $O(m \log m)$ time.*

PROOF. Consider a node u in the checkpoint tree with children v and w whose subtree contains m updates. The time-fusible priority queue containing all updates in u's subtree can be computed in $O(m)$ time from the 2 time-fusible priority queues associated with v and w. First the FUSE operation outlined in Section 9.2 is performed to merge v and w. The resulting time-fusible priority queue may represent Q_{now} and Q_{del} using multiple trees, but these trees can be merged in $O(m)$ time. Using this merge procedure, a subtree of the checkpoint tree is rebuilt by first placing all m updates at the leaves of a balanced tree, and then performing merges from the leaves upward. Each update is involved in $O(\log m)$ merges, so the total time to rebuild the subtree is $O(m \log m)$. □

A more efficient implementation of the FIND-DELETION-TIME(k) operation can be obtained by performing a binary search directly on the checkpoint tree. The high-level idea is to perform a binary search for the time of deletion by keeping track of the current number of surviving keys that are less than or equal to k at any particular time. This result is stated in Lemma 57 without proof.

Lemma 57 *The* FIND-DELETION-TIME *operation which performs a binary search directly on the checkpoint tree data structure runs in $O(\log^2 m)$ time.*

Theorem 58 *The fully retroactive priority queue performs updates in $O(\log^2 m)$ amortized time when using a checkpoint tree with the memoized subtree rebuilding procedure, and performs* FIND-DELETION-TIME *operations in $O(\log^2 m)$ time.*

PROOF. We may improve the runtime of update operations given in Theorem 55 to $O(\log^2 m)$ amortized time by using the rebuild procedure given in Lemma 56. We still perform update operations by inserting or deleting the update from all the partially retroactive structures along our search path, as in Lemma 48. However, we provide a tighter analysis for rebalancing the tree.

Since our scapegoat tree has balance factor, $\alpha = \frac{9}{10}$, we perform $\Omega(m)$ operations before rebuilding a subtree. Each update operation may charge $\log m$ to every node along the search path. Therefore, each update operation creates $O(\log^2 m)$ charges. Rebuilding a subtree of size m takes $O(m \log m)$ time as given in Lemma 56. We perform at most $O(\log m)$ rebuilds. Each node that is rebuilt has a stored charge of $O(m \log m)$. Therefore, the time of rebuild may be paid for by the stored charge. The time per update operation is then $O(\log^2 m)$. □

9.4.3 Faster Find-Deletion-Time queries

Next we discuss how to implement Find-Deletion-Time in $O(\log^2 m)$ time by performing a search directly on the checkpoint tree data structure. In this section, for ease of explanation, we will use a counter, d_i, to keep track of the number of Delete-Min operations performed on the partially retroactive structure in node i.

Given a key k, Find-Deletion-Time(k) returns the deletion time, d_k, of the element or ∞ if the element was never deleted or never inserted into the queue. By inspection, an element was ever inserted into a partially retroactive priority queue if and only if it is present in either the Q_{now} or Q_{del} of the queue. Therefore, an element was inserted into the fully retroactive priority queue if and only if it is present in the partially retroactive priority queue at its root. Our procedure first checks the root to verify that the element was inserted and deleted; it returns ∞ otherwise.

For the purposes of this operation, we assume that all keys inserted into the queue are unique. First, we check whether k is in the Q_{del} contained in the root of the fully retroactive tree. If k is in the root Q_{del}, then it was deleted at some time before the present, and we proceed with the following recursive procedure to find t_k, the time of deletion of key k. If it is not in Q_{del}, then it was either never deleted or never inserted, and we return ∞.

We define s to be the number of surviving elements with keys less than or equal to k, c to be the current node (at the very first step of the recursive procedure c is the root), c_l to be the left child of c, c_r to be the right child of c, c_{gl} to be the left child of c_r, and c_{gr} to be the right child of c_r. If a child does not exist, then the value of the corresponding variable is set to `null`.

The key concept behind the Find-Deletion-Time(k) algorithm is to binary search for the time of deletion by keeping track of the current number of surviving elements that are less than or equal to k at any particular time. If this counter becomes zero, then we have deleted k. We define this counter as s. To help us in the detailed explanation of this algorithm in Figure 9-3, we define w_i to be the number of elements in Q_{now} of node i that are at most k, $w_i = |\{k' : k' \in Q_{i,now} \cap k' \leq k\}|$, v_i to be the number of elements in Q_{del} of node i that are at most k, $v_i = |\{k' : k' \in Q_{i,del} \cap k' \leq k\}|$, and $s'_i = s + v_i - d_i$ of node i. Each time we check a node i, we check its s'_i which tells us either that k was deleted in the subtree rooted at i or it was not. If $s'_i \leq 0$, then an update contained in i deleted k and we binary search for the precise update. Otherwise, we can conclude that none of the updates in the subtree rooted at i deleted k, and so, we do not need to search any more nodes in the subtree.

Acknowledgments

This research was initiated during the open-problem sessions of the MIT class 6.851: Advanced Data Structures taught by E. Demaine in Spring 2014. We thank Adam Hesterberg, Ofir Nachum, and other members of the class for helpful discussion regarding this problem.

```
FIND-DELETION-TIME($k$)
1. $c$ is the current node. Set $c$ to the root if $c$ has not yet been set. Let $s = 0$.
2. If $k \in Q_{now}$ of $c$:
    1. If $c$ is the root, return $\infty$.
    2. Else, set $s = \max(0, s'_c) + w_c$. Update $c$ to be $c_{gl}$ if it exists. If $c_r$ only contains one
       update, then that update deletes $k$; return the time of that update. Otherwise, go back
       to Step 2.
3. Else if $k \in Q_{del}$ of $c$:
    1. Update $c$ to be $c_l$.
    2. Go back to Step 2.
3. Else ($k \notin Q_{del}$ and $k \notin Q_{now}$):
    1. If $k \in Q_{now}$ was found at a previous step:
        1. If $s'_c \leq 0$: check to see if there is only one update within the subtree rooted at $c$.
           If there is only one update, return the time, $t$, of the update. If there is more than
           one update, set $c$ to be $c_l$.
        2. If $s'_c > 0$: set $s = s'_c + w_c$. Set $c$ to be $c_{gl}$ if it exists. If $c_{gl}$ does not exist, then $c_r$
           contains only one update and that update deletes $k$; return the time of the update.
        3. Go back to Step 2.
    4. Else ($k$ was not found in previous step):
        1. Set $s = \max(0, s'_c) + w_c$.
        2. Set $c$ to be $c_{gl}$.
        3. Go back to Step 2.
```

Figure 9-3: Pseudocode for the FIND-DELETION-TIME(k) operation.

Chapter 10

Conclusion

10.1 Summary

This thesis has shown that the complexity of parallel programming can be reduced by developing programming technologies that facilitate the development of quality code that has simple understandable structure and perform well in practice. There were three categories of programming technologies developed in the artifacts of this thesis: shared-memory multicore algorithms, multicore-centric systems for scientific computing, and tools for measuring and optimizing anomalous, or worst-case, behavior in parallel systems.

In Chapters 2–4, I discussed artifacts that developed shared-memory multicore algorithms that are deterministic and have the semantics of serial code. The Chromatic artifact in Chapter 2 showed how dynamic data-graph computations can be executed deterministically and with serial semantics by using chromatic scheduling. The Color artifact in Chapter 3 demonstrated that vertex-ordering heuristics can be used in parallel greedy graph coloring algorithms to compute the same result as the serial algorithm using the same ordering. The Color artifact showed how the parallelism of the greedy algorithm can be theoretically analyzed, and how the historically useful serial ordering heuristics can be naturally coarsened to increase theoretical parallelism. The PARAD artifact in Chapter 4 provided a work-efficient and parallelism preserving algorithm for performing automatic reverse-mode differentiation of parallel programs — extending the commonly used serial program transformation. The results in these artifacts demonstrated that the complexity of shared-memory multicore programming can be reduced, in many cases, by using parallel algorithms that have the semantics of serial code without compromising theoretical scalability or real-world performance.

In Chapters 5–6, I discussed artifacts that demonstrated the effectiveness of multicore-centric computing systems for scientific computing in the field of connectomics. The Connectomics artifact in Chapter 5 demonstrated that a mixture of traditional performance optimizations and the application of quality parallel algorithms can allow a single large multicore to outperform more complex systems employing distributed computing and GPUs. The Alignment artifact in Chapter 6 showed how relatively simple algorithm design techniques and careful pipeline design can enable a multicore-centric system to scale to larger input sizes and scale horizontally over multiple machines in the cluster with low overhead. The results in these artifacts demonstrated that well-designed multicore-centric systems can be highly effective at solving problems in scientific computing, even out-performing systems using GPUs and distributed computing.

In Chapters 7–8, I discussed artifacts that related to the analysis and optimization of anomalous, or worst-case, behavior in parallel systems. The Cilkmem artifact in Chapter 7 provided theoretically efficient algorithms and implementations for measuring the exact and approximate p-processor memory high-water mark in fork-join parallel programs. The Cilkmem tool, which implements these algorithms, allows practitioners to measure the maximum memory their program requires in a worst case execution. The Reissue artifact in Chapter 8 illustrated the effectiveness of the simple single-time random-reissue (SINGLER) policy family in practice and in theory, using a simplified analytical model to compare the relative power of differently parametrized reissue policy families. The results in these artifacts demonstrated the ability of principled tools to analyze and optimize application-specific performance objectives in ways that have theoretical guarantees and are effective in practice.

10.2 Taming complexity in a post-Moore's-law world

An examination of current trends in the semiconductor industry foretells a significant rise in the importance of software performance engineering.

The historical decoupling of application logic from performance.

In the early 2000s, computer hardware engineers were able to offer a simple and compelling value proposition to industry and the other sciences: focus on the logic of your domain-specific application, and rely on the seemingly inexorable march of microprocessor improvements to solve your performance and scaling problems.

The end of Dennard scaling in 2005, however, marked the end of this simple argument. As Dennard scaling ended, the semiconductor industry turned towards the design of multicore hardware which could be predictably improved over time by relying on technological advances to increase transistor density. Thus, the value proposition was amended: if you design your application to have copious amounts of logical parallelism, the semiconductor industry will provide you with predictable performance improvements over time.

By 2016, the semiconductor industry acknowledged that increases in transistor density had plateaued. The International Technology Roadmap for Semiconductors announced a strategy called "More than Moore" that turned to focus on the needs of applications to drive chip development instead of focusing on further scaling of semiconductor technology for general purpose computing. The simple value proposition between the semiconductor industry and users of computing was now dead. Going forward, the engineering of high performance computing technology and domain-specific program logic would be intertwined.

The increased complexity of programming modern hardware has made it more difficult for even expert programmers to develop high-quality software systems. For the scientific computing community, developing quality software systems that can take advantage of the latest supercomputing technology has become a costly endeavour: requiring more time, more expensive hardware, more expert programmers, and more complex software systems.

Building a toolbox of principled technologies for average programmers

I believe that we can make it easier for average programmers to tap into the power provided by new hardware systems by developing a toolbox of principled programming technologies.

Developing quality software for scientific computing depends on the availability of performant algorithms, data structures, software libraries, and system design principles. The development of such programming technologies that are simple to use, theoretically sound, and performant can allow programmers to focus on their application-specific logic.

The artifacts in this thesis presented several programming technologies for the shared-memory multicore platform. Of course, the programming technologies presented in this thesis only address a small subset of the challenges involved in general parallel programming. What this thesis demonstrates, however, is that the pursuit of simple, theoretically sound, and performant programming technologies can succeed in reducing the complexity of programming parallel systems.

Bibliography

[1] L. Adams and J. Ortega. A multi-color SOR method for parallel computation. In *ICPP*, 1982.

[2] Pankaj K Agarwal. Range searching. Technical report, DTIC Document, 1996.

[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification Version 1.0*. Sun Microsystems, Inc., March 2008.

[4] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A comparison of parallel graph coloring algorithms. Technical report, Northeast Parallel Architecture Center, Syracuse University, 1995.

[5] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, December 1986.

[6] Amazon. Amazon elastic file system. https://aws.amazon.com/efs/, 2019.

[7] David G Andersen, Hari Balakrishnan, M Frans Kaashoek, and Rohit N Rao. Improving web availability for clients with monet. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 115–128. USENIX Association, 2005.

[8] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(5):898–916, 2011.

[9] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 560–569. IEEE, 1996.

[10] Esther M Arkin and Ellen B Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 1987.

[11] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 119–129, 1998.

[12] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[13] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[14] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[15] Leonid Barenboim and Michael Elkin. Distributed (Δ+1)-coloring in linear (in Δ) time. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 111–120, New York, NY, USA, 2009. ACM.

[16] Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

[17] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[18] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):55955637, January 2017.

[19] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 399–409. IEEE, 2000.

[20] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 133–144, 2004.

[21] J. L. Bentley. Solutions to klees rectangle problems. Technical report, Carnegie Mellon University,, 1977.

[22] Stefan Berchtold, Christian Böhm, Bernhard Braunmüller, Daniel A Keim, and Hans-Peter Kriegel. Fast parallel similarity search in multimedia databases. In *ACM SIGMOD Int. Conf. on Management of Data*, 1997.

[23] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. *SIGPLAN Not.*, 45(3):53–64, March 2010.

[24] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, Orlando, Florida, USA, 2009. ACM.

[25] Gilles Bertrand and Zouina Aktouf. Three-dimensional thinning algorithm using subfields. volume 2356, pages 113–124, 1995.

[26] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

[27] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[28] Christian Bischof, Niels Guertler, Andreas Kowarz, and Andrea Walther. Parallel reverse mode automatic differentiation for openmp programs with adol-c. In *Advances in Automatic Differentiation*, pages 163–173, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[29] Christian H. Bischof. Issues in parallel automatic differentiation. In *Proceedings of the 1991 International Conference on Supercomputing*, pages 146–153. ACM Press, 1991.

[30] Guy E. Blelloch. Prefix sums and their applications. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[31] Guy E. Blelloch. NESL: A nested data-parallel language. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[32] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996.

[33] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. *SIGPLAN Not.*, 47(8):181–192, February 2012.

[34] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM SPAA*, 2012.

[35] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM.

[36] Robert D. Blumofe, Matteo Frigo, Chrisopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[37] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[38] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[39] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 356–368, November 1994.

[40] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, February 1998.

[41] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[42] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, University of Texas at Austin, 1998.

[43] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[44] Léon. Bottou, Frank E. Curtis, and Jorge. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[45] Peter J Braam and Rumi Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[46] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: Composable transformations of Python+NumPy programs, 2018.

[47] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[48] Daniel Brélaz. New methods to color the vertices of a graph. *CACM*, 1979.

[49] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[50] Preston Briggs. *Register allocation via graph coloring.* PhD thesis, Rice University, 1992.

[51] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

[52] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, WADS '99, pages 37–48, London, UK, UK, 1999. Springer-Verlag.

[53] Mark R Brown and Robert E Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM journal on computing*, 9(3):594–614, 1980.

[54] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition, 2015.

[55] H. Martin Bücker, Bruno Lang, Dieter an Mey, and Christian H. Bischof. Bringing together automatic differentiation and openmp. In *Proceedings of the 15th International Conference on Supercomputing*, ICS 01, page 246251, New York, NY, USA, 2001. Association for Computing Machinery.

[56] R.L. Burden, J.D. Faires, and A.M. Burden. *Numerical Analysis*. Cengage Learning, 2015.

[57] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[58] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM.

[59] K Cameron and J Edmonds. Algorithms for optimal anti-chains. In *Research report CORR 79*, volume 22. Department of Combinatorics and Optimization. University of Waterloo , 1979.

[60] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*, September 2008.

[61] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[62] Ümit V. Çatalyürek, John Feo, Assefaw Hadish Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for muti-core and massively multithreaded architectures. *CoRR*, 2012.

[63] Gregory J Chaitin. Register allocation & spilling via graph coloring. In *ACM SIGPLAN Notices*, 1982.

[64] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 1981.

[65] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*. SIAM, 2004.

[66] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Trans. Softw. Eng.*, 26(3):197–211, March 2000.

[67] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[68] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. Association for Computing Machinery.

[69] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[70] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[71] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 571–582, Berkeley, CA, USA, 2014. USENIX Association.

[72] Soumith Chintala. Convnet benchmarks. https://github.com/soumith/convnet-benchmarks.

[73] Dan Ciresan, Alessandro Giusti, Luca M Gambardella, and Jürgen Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In *Advances in neural information processing systems*, pages 2843–2851, 2012.

[74] R Cole and U Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 206–219, New York, NY, USA, 1986. ACM.

[75] T. Coleman and J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, 1983.

[76] Melvin E. Conway. A multiprocessor system design. In *AFIPS*, 1963.

[77] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[78] Intel Corporation. Intel math kernel library, 2016.

[79] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, October 1971.

[80] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 4:1–4:16, New York, NY, USA, 2016. ACM.

[81] Joseph C. Culberson. Iterated greedy graph coloring and the difficulty landscape. Technical report, University of Alberta, 1992.

[82] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, December 2008.

[83] Inc. DataStax. Datastax distribution of apache cassandra 3.x, 2016.

[84] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[85] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[86] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[87] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[88] Erik D Demaine, John Iacono, and Stefan Langerman. Retroactive data structures, 2007. Originally in SODA 2003.

[89] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[90] J E Dennis, Jr. and Trond Steihaug. On the successive projections approach to least-squares problems. *SIAM J. Numer. Anal.*, 23(4):717–733, August 1986.

[91] A Descampe, F Devaux, H Drolon, D Janssens, and Y Verschueren. Openjpeg 2.0. 0, 2012.

[92] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. *SIGPLAN Not.*, 44(3):85–96, March 2009.

[93] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: A relaxed consistency deterministic computer. *SIGPLAN Not.*, 47(4):67–78, March 2011.

[94] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining numa locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM.

[95] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[96] Krzysztof Diks. A fast parallel algorithm for six-colouring of planar graphs. In *Mathematical Foundations of Computer Science*. 1986.

[97] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. Race detection in two dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 101–110, Portland, Oregon, USA, 2015. ACM.

[98] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. In *STOC*, 1993.

[99] Derek L Eager, John Zahorjan, and Edward D Lazowska. Speedup versus efficiency in parallel systems. *IEEE transactions on computers*, 38(3):408–423, 1989.

[100] AL Eberle, S Mikula, R Schalek, J Lichtman, ML KNOTHE TATE, and D Zeidler. High-resolution, high-throughput imaging with a multibeam scanning electron microscope. *Journal of microscopy*, 259(2):114–120, 2015.

[101] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[102] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: Scalable nonzero indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 13–22, New York, NY, USA, 2007. ACM.

[103] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. Studying overheads in massively parallel min/max-tree evaluation. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–103, June 1994.

[104] Linqing Feng, Ting Zhao, and Jinhyun Kim. neuTube 1.0: a New Design for Efficient Neuron Reconstruction Software Based on the SWC Format. *eneuro*, January 2015.

[105] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[106] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, June 1997.

[107] Jeremy T. Fineman and Charles E. Leiserson. Race detectors for Cilk and Cilk++ programs. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1706–1719. Springer, 2011.

[108] Raphael Finkel and Udi Manber. DIB — A distributed implementation of backtracking. *ACM TOPLAS*, 9(2):235–256, April 1987.

[109] Matteo Fischetti, Silvano Martello, and Paolo Toth. The fixed job schedule problem with spread-time constraints. *Operations Research*, 1987.

[110] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[111] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 159–170. ACM, 2013.

[112] Wikimedia Foundation. Wikipedia: Database, 2016.

[113] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in $X+Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24(2):197–208, 1982.

[114] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

[115] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[116] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[117] Igal Galperin and Ronald L Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1993.

[118] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. Reducing latency via redundant requests: Exact analysis. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 347–360. ACM, 2015.

[119] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[120] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1976.

[121] Assefaw H. Gebremedhin, Duc Nguyen, Md. Mostofa Ali Patwary, and Alex Pothen. ColPack: Software for graph coloring and related problems in scientific computing. *ACM Trans. on Mathematical Software*, 2013.

[122] Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 2000.

[123] Alan E. Gelfand and Adrian F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85(410):398–409, June 1990.

[124] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741, November 1984.

[125] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 158–168, New York, NY, USA, 1989. ACM.

[126] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in Matlab: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, January 1992.

[127] Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms (TALG)*, 5(3):28, 2009.

[128] Alessandro Giusti, Dan Claudiu Ciresan, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Fast image scanning with deep max-pooling convolutional neural networks. In *ICIP*, page in press, 2013.

[129] Robert K. Gjertsen Jr., Mark T. Jones, and Paul E. Plassmann. Parallel heuristics for improved, balanced graph colorings. *JPDC*, 1996.

[130] Sébastien Godard. Sysstat: System performance tools for the linux os, 2004.

[131] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discret. Math.*, 1(4):434–446, October 1988.

[132] Mark Goldberg and Thomas Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 18(2):419–427, 1989.

[133] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, 2(2):205–224, 1965.

[134] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[135] Google. Google cloud platform blog: Google supercharges machine learning tasks with tpu custom chip, 2016.

[136] Google. Protocol buffers. https://developers.google.com/protocol-buffers/, 2019.

[137] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[138] Felix Gremse, Andreas Höfter, Lukas Razik, Fabian Kiessling, and Uwe Naumann. Gpu-accelerated adjoint algorithmic differentiation. *Computer Physics Communications*, 200:300–311, 2016.

[139] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[140] Yan Gu, Julian Shun, Yihan Sun, and Guy E Blelloch. A top-down parallel semisort. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 24–34, 2015.

[141] Leo J Guibas, Edward M McCreight, Michael F Plass, and Janet R Roberts. A new representation for linear lists. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60. ACM, 1977.

[142] Robert H. Halstead, Jr. Implementation of MultiLisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[143] Robert H. Halstead, Jr. MultiLisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

[144] James Hamilton. The cost of latency, 2009.

[145] Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3), May 2013.

[146] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 166–177, New York, NY, USA, 2014. ACM.

[147] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 27–40, New York, NY, USA, 2015. ACM.

[148] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *ACM Symposium on Cloud Computing (SOCC)*, page 12, 2012.

[149] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview scalability analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 145–156, New York, NY, USA, 2010. ACM.

[150] DP John Hennessy. A conversation with john hennessy and david patterson. *ACM Queue*, 4(10), 2006.

[151] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[152] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pnfs. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 18–27. IEEE, 2005.

[153] F. L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematical Physics*, 1927.

[154] M. Hitz, J. Grabmeier, E. Kaltofen, and V. Weispfenning. *Computer Algebra Handbook: Foundations · Applications · Systems.* Springer Berlin Heidelberg, 2012.

[155] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[156] Robin J Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):26, 2014.

[157] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM.

[158] P. Hovland and C. Bischof. Automatic differentiation for message-passing parallel programs. In *IPPS*, pages 98–104, March 1998.

[159] Paul D. Hovland, Christian H. Bischof, and Lucas Roh. Automatic differentiation of a parallel molecular dynamics application. In *PPSC*. SIAM, 1997.

[160] Derek R. Hower, Polina Dudnik, Mark D. Hill, and David A. Wood. Calvin: Deterministic or not? free will to choose. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 333–334, Washington, DC, USA, 2011. IEEE Computer Society.

[161] Jan Hückelheim, Paul Hovland, Michelle Mills Strout, and Jens-Dominik Mller. Reverse-mode algorithmic differentiation of an openmp-parallel compressible flow solver. *The International Journal of High Performance Computing Applications*, 33(1):140–154, 2019.

[162] Jan Hückelheim and Paul D. Hovland. Automatic differentiation of parallelised convolutional neural networks - lessons from adjoint pde solvers. In *NIPS*, 2017.

[163] Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. Automatic differentiation for adjoint stencil loops. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[164] Jan Christian Hückelheim, Paul D. Hovland, Michelle Mills Strout, and Jens-Dominik Müller. Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation. *Optimization Methods and Software*, 33:672–693, 2018.

[165] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta informatica*, 17(2):157–184, 1982.

[166] IBM. Introducing a brain-inspired computer, 2016.

[167] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3(25):602, 2018.

[168] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Standard 1003.1, 1996 Edition.

[169] Intel. The Threading Building Blocks. http://software.intel.com, 2012.

[170] Intel. Intel Cilk Plus. http://software.intel.com, 2013.

[171] Intel. Intel Cilk Plus. Available from http://software.intel.com, 2013.

[172] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf.

[173] Intel Corporation. Intel VTune Amplifier. https://software.intel.com/en-us/vtune, 2019.

[174] itseez. Open source computer vision library, 2016.

[175] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.

[176] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. *ACM SIGCOMM Computer Communication Review*, 43(4):219–230, 2013.

[177] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[178] Maximilian Joesch, David Mankus, Masahito Yamagata, Ali Shahbazi, Richard Schalek, Adi Suissa-Peleg, Markus Meister, Jeff W Lichtman, Walter J Scheirer, and Joshua R Sanes. Reconstruction of genetically identified neurons imaged by serial-section electron microscopy. *elife*, 5:e15015, 2016.

[179] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, May 1993.

[180] Mark T Jones and Paul E Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 1994.

[181] Gauri Joshi, Emina Soljanin, and Gregory Wornell. Efficient redundancy techniques for latency reduction in cloud systems. *arXiv preprint arXiv:1508.03599*, 2015.

[182] Gauri Joshi, Emina Soljanin, and Gregory Wornell. Queues with redundancy: Latency-cost analysis. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):54–56, 2015.

[183] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. *Transactions on Parallel Computing*, 3(1):2:1–2:31, 2016.

[184] Tim Kaler, William Hasenplaugh, Tao B Schardl, and Charles E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. In *SPAA*, 2014.

[185] Tim Kaler, Yuxiong He, and Sameh Elnikety. Optimal reissue policies for reducing tail latency. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 195–206, 2017.

[186] Tim Kaler, William Kuszmaul, Tao B Schardl, and Daniele Vettorel. Cilkmem: Algorithms for analyzing the memory high-water mark of fork-join parallel programs. In *Symposium on Algorithmic Principles of Computer Systems*, pages 162–176. SIAM, 2020.

[187] Tim Kaler, Brian Wheatman, and Sarah Wooders. High-throughput image alignment for connectomics using frugal snap judgments: Poster. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 433–434, New York, NY, USA, 2019. ACM.

[188] Tim Kaler, Brian Wheatman, and Sarah Wooders. High-throughput image alignment for connectomics using frugal snap judgments. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020.

[189] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *JACM*, 40(3):765–789, July 1993.

[190] Narayanan Kasthuri, Ken Hayworth, Juan C Tapia, Richard Schalek, S Nundy, and Jeff W Lichtman. The brain on tape: Imaging an ultra-thin section library (utsl). In *Soc. Neurosci. Abstr*, 2009.

[191] Narayanan Kasthuri, Kenneth Jeffrey Hayworth, Daniel Raimund Berger, Richard Lee Schalek, José Angel Conchello, Seymour Knowles-Barley, Dongil Lee, Amelio Vázquez-Reina, Verena Kaynig, Thouis Raymond Jones, et al. Saturated reconstruction of a volume of neocortex. *Cell*, 162(3):648–661, 2015.

[192] Verena Kaynig, Amelio Vazquez-Reina, Seymour Knowles-Barley, Mike Roberts, Thouis R Jones, Narayanan Kasthuri, Eric Miller, Jeff Lichtman, and Hanspeter Pfister. Large-scale automatic reconstruction of neuronal processes from electron microscopy images. *Medical image analysis*, 22(1):77–88, 2015.

[193] Gershon Kedem. Automatic differentiation of computer programs. Technical report, WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER, 1976.

[194] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 7–16, New York, NY, USA, 2015. ACM.

[195] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[196] Seymour Knowles-Barley. Rhoana git. `https://github.com/Rhoana/membrane_cnn/tree/master/maxout`.

[197] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* MIT Press, Cambridge, MA, USA, 1994.

[198] S Rao Kosaraju. Localized search in sorted lists. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 62–69. ACM, 1981.

[199] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, June 1989.

[200] Fabian Kuhn. Weak graph colorings: Distributed algorithms and applications. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 138–144, New York, NY, USA, 2009. ACM.

[201] Fabian Kuhn and Rogert Wattenhofer. On the complexity of distributed graph coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 7–15, New York, NY, USA, 2006. ACM.

[202] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-645.

[203] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[204] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[205] Cliff Lasser and Steve M. Omohundro. The essential Lisp manual. Technical report, Thinking Machines, Cambridge, MA USA, 1986.

[206] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002. *See* `http://llvm.cs.uiuc.edu`.

[207] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, page 75, Palo Alto, California, March 2004.

[208] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[209] Jonathan I Leckenby, Miranda A Chacon, Adriaan O Grobbelaar, and Jeff W Lichtman. Imaging peripheral nerve regeneration: A new technique for 3d visualization of axonal behavior. *Journal of Surgical Research*, 242:207–213, 2019.

[210] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20, 2015.

[211] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[212] I-Ting Angelina Lee. *Memory Abstractions for Parallel Programming.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, 2012.

[213] I-Ting Angelina Lee, Aamir Shafi, and Charles E. Leiserson. Memory-mapping support for reducer hyperobjects. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 287–297, New York, NY, USA, 2012. ACM.

[214] Kangwook Lee, Ramtin Pedarsani, and Kannan Ramchandran. On scheduling redundant requests with cancellation overheads. In *Proc. of the 53rd Annual Allerton conference on Communication, Control, and Computing*, 2015.

[215] Kisuk Lee, Aleksandar Zlateski, Vishwanathan Ashwin, and H Sebastian Seung. Recursive Training of 2D-3D Convolutional Networks for Neuronal Boundary Prediction. In *Advances in Neural Information Processing Systems*, pages 3559–3567, 2015.

[216] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[217] Wei-Chung Allen Lee, Vincent Bonin, Michael Reed, Brett J Graham, Greg Hood, Katie Glattfelder, and R Clay Reid. Anatomy and function of an excitatory network in the visual cortex. *Nature*, 532(7599):370–374, 2016.

[218] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine.*

[219] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, 2010.

[220] J. Leskovec. SNAP: Stanford network analysis platform. http://snap.stanford.edu/data/index.html, 2013.

[221] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–es, 2007.

[222] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[223] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[224] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. Malt: Distributed data-parallelism for existing ML applications. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 3:1–3:16, New York, NY, USA, 2015. ACM.

[225] Yawei Li and Zhiling Lan. Exploit failure prediction for adaptive fault-tolerance in cluster computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 8–pp. IEEE, 2006.

[226] Jeff W Lichtman and Winfried Denk. The big and the small: challenges of imaging the brains circuits. *Science*, 334(6056):618–623, 2011.

[227] Jeff W Lichtman, Hanspeter Pfister, and Nir Shavit. The big data challenges of connectomics. *Nature neuroscience*, 17(11):1448–1454, 2014.

[228] Jeff W. Lichtman and Joshua R. Sanes. Ome sweet ome: what can the genome tell us about the connectome? *Current Opinion in Neurobiology*, 18(3):346–353, June 2008.

[229] Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, February 1992.

[230] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, Jun 1976.

[231] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[232] L. Lovász, M. Saks, and W. T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Math.*, 1989.

[233] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[234] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[235] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[236] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 1986.

[237] Apache Lucene. Apache lucene, 2010.

[238] George S Lueker. A data structure for orthogonal range queries. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 28–34. IEEE, 1978.

[239] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[240] David Maier and Sharon C Salveter. Hysterical b-trees. *Information Processing Letters*, 12(4):199–202, 1981.

[241] Jeremy Maitin-Shepard, Viren Jain, Michal Januszewski, Peter Li, Jörgen Kornfeld, Julia Buhmann, and Pieter Abbeel. Combinatorial energy learning for image segmentation. *arXiv preprint arXiv:1506.04304*, 2015.

[242] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[243] Loris Marchal, Hanna Nagy, Bertrand Simon, and Frédéric Vivien. Parallel scheduling of dags under memory constraints. In *IPDPS*, pages 204–213. IEEE, 2018.

[244] Dániel Marx. Graph colouring problems and their applications in scheduling. *John von Neumann Ph.D. Students Conf.*, 2004.

[245] Jonathan Masci, Alessandro Giusti, Dan Ciresan, Gabriel Fricout, and Jurgen Schmidhuber. A fast learning algorithm for image segmentation with max-pooling convolutional networks. In *Image Processing (ICIP), 2013 20th IEEE International Conference on*, pages 2713–2717. IEEE, 2013.

[246] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *JACM*, 1983.

[247] Alexander Matveev, Yaron Meirovitch, Hayk Saribekyan, Wiktor Jakubiuk, Tim Kaler, Gergely Odor, David Budden, Aleksandar Zlateski, and Nir Shavit. A multicore path to connectomics-on-demand. In *PPoPP*, pages 267–281, 2017.

[248] Andrew McCallum. Cora data set. http://people.cs.umass.edu/mccallum/data.html, 2012.

[249] Mike McCandless. Lucene nightly benchmarks, 2010.

[250] Michael D McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 5–5. USENIX Association, 2010.

[251] D. McGrady. Avoiding contention using combinable objects. *Microsoft Developer Network*, 2008.

[252] Marina Meilă. Comparing clusteringsan information based distance. *Journal of multivariate analysis*, 98(5):873–895, 2007.

[253] Marina Meilă. Comparing clusteringsan information based distance. *Journal of multivariate analysis*, 98(5):873–895, 2007.

[254] Y. Meirovitch, A. Matveev, H. Saribekyan, D. Budden, D. Rolnick, G. Odor, S. K.-B. T. R. Jones, H. Pfister, J. W. Lichtman, and N. Shavit. A Multi-Pass Approach to Large-Scale Connectomics. *ArXiv e-prints*, December 2016.

[255] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[256] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. Web data commons — hyperlink graphs.

[257] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the Web — analyzed on different aggregation levels. *Journal of Web Science*, 1(1):33–47, 2015.

[258] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS '11)*, October 2011.

[259] Tom Mitchell. NPIC500 data set. http://www.cs.cmu.edu/tom/10709_fall2009/NPIC500.pdf, 2009.

[260] John Mitchem. On various algorithms for estimating the chromatic number of a graph. *The Computer Journal*, 1976.

[261] Alina N. Moga, Bogdan Cramariuc, and Moncef Gabbouj. Parallel watershed transformation algorithms for image segmentation. *Parallel Comput.*, 24(14):1981–2001, December 1998.

[262] Josh Lyskowski Morgan, Daniel Raimund Berger, Arthur Willis Wetzel, and Jeff William Lichtman. The fuzzy logic of network connectivity in mouse visual thalamus. *Cell*, 165(1):192–206, 2016.

[263] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, pages 467–475, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[264] Nervana. Neon. https://github.com/NervanaSystems/neon.

[265] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[266] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[267] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[268] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: Ondemand, portable and parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 499–512, New York, NY, USA, 2014. ACM.

[269] Kamal Nigam and Rayid Ghani. Analyzing the effectiveness and applicability of cotraining. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, pages 86–93, New York, NY, USA, 2000. ACM.

[270] Rishiyur S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[271] Juan Nunez-Iglesias, Ryan Kennedy, Toufiq Parag, Jianbo Shi, and Dmitri B Chklovskii. Machine learning of hierarchical clustering to segment 2D and 3D images. *PloS one*, 8(8):e71715, 2013.

[272] Juan Nunez-Iglesias, Ryan Kennedy, Stephen M Plaza, Anirban Chakraborty, and William T Katz. Graph-based active learning of agglomeration (gala): a python library to segment 2d and 3d neuroimages. *Frontiers in neuroinformatics*, 8, 2014.

[273] NVIDIA. Nvidia cudnn - gpu accelerated deep learning, 2016.

[274] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. *SIGARCH Comput. Archit. News*, 37(1):97–108, March 2009.

[275] *OpenMP Application Program Interface, Version 3.0*, May 2008.

[276] Oracle. Sun studio 12: Performance analyzer. Available at https://docs.oracle.com/cd/E19205-01/819-5264/, 2010.

[277] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA*, 1984.

[278] Toufiq Parag, Anirban Chakraborty, Stephen Plaza, and Louis Scheffer. A context-aware delayed agglomeration framework for electron microscopy segmentation. *PloS one*, 10(5):e0125825, 2015.

[279] Toufiq Parag, Anirban Chakrobarty, and Stephen Plaza. A context-aware delayed agglomeration framework for em segmentation. *CoRR*, 2014.

[280] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[281] Suhas S. Patil. Record of the project MAC conference on concurrent systems and parallel computation. chapter Closure Properties of Interconnections of Determinate Systems, pages 107–116. ACM, New York, NY, USA, 1970.

[282] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[283] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of parallelism in algorithms. In *ACM PLDI*, 2011.

[284] Stephen M Plaza and Stuart E Berg. Large-scale electron microscopy image segmentation in spark. *arXiv preprint arXiv:1604.00385*, 2016.

[285] Daniel Quinlan and Michael Kerrisk. proc — process information pseudo-filesystem. Available at http://man7.org/linux/man-pages/man5/proc.5.html, 2017.

[286] Louis B Rall and George F Corliss. An introduction to automatic differentiation. *Computational Differentiation: Techniques, Applications, and Tools*, 89, 1996.

[287] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 368–383. Springer Berlin / Heidelberg, 2010.

[288] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, 2012.

[289] Santiago Ramón and S Cajal. *Textura del Sistema Nervioso del Hombre y de los Vertebrados*, volume 2. Madrid Nicolas Moya, 1904.

[290] A. Rauh and G.R. Arce. A fast weighted median algorithm based on quickselect. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 105–108, Sept 2010.

[291] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[292] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, et al. Interactive supercomputing on 40,000 cores for machine learning and data analysis. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[293] Rhoana. Fijibento, 2018.

[294] William R Gray Roncal, Dean M Kleissas, Joshua T Vogelstein, Priya Manavalan, Kunal Lillaney, Michael Pekala, Randal Burns, R Jacob Vogelstein, Carey E Priebe, Mark A Chevillet, et al. An automated images-to-graphs framework for high resolution connectomics. *Frontiers in neuroinformatics*, 9, 2015.

[295] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015*, pages 234–241. Springer, 2015.

[296] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. Ieee, 2011.

[297] Youcef Saad. *SPARSKIT: A basic toolkit for sparse matrix computations*. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990.

[298] Stephan Saalfeld, Albert Cardona, Volker Hartenstein, and Pavel Tomančák. As-rigid-as-possible mosaicking and serial section registration of large sstem datasets. *Bioinformatics*, 26(12):i57–i63, 2010.

[299] Punam K. Saha, Gunilla Borgefors, and Gabriella Sanniti di Baja. A survey on skeletonization algorithms and their applications. *Pattern Recognition Letters*, 76:3–12, June 2016.

[300] A.E. Sariyuce, E. Saule, and U.V. Catalyurek. Improving graph coloring on distributed-memory parallel computers. In *HiPC*, 2011.

[301] Michel Schanen, Uwe Naumann, Laurent Hascoët, and Jean Utke. Interpretative adjoints for numerical simulation codes using mpi. *Procedia Computer Science*, 1(1):1825 – 1833, 2010. ICCS 2010.

[302] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. The CSI framework for compiler-inserted program instrumentation. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(2):43:1–43:25, December 2017.

[303] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. The Cilkprof scalability profiler. In *SPAA*, pages 89–100, 2015.

[304] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *PPoPP*, pages 249–265, 2017.

[305] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding recursive fork-join parallelism into LLVM's intermediate representation. *ACM Trans. Parallel Comput.*, 6(4), December 2019.

[306] Louis Scheffer, Bill Karsh, and Shiv Vitaladevun. Automated alignment of imperfect em images for neural reconstruction. abs/1304.6034, 04 2013.

[307] Peter Schiffer and Michael Kerrisk. memusage — profile memory usage of a program. Available at http://man7.org/linux/man-pages/man1/memusage.1.html, 2014.

[308] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Conference*, 2009.

[309] Sebastian Seung. *Connectome: How the brain's wiring makes us who we are.* Houghton Mifflin Harcourt, 2012.

[310] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. The mds queue: Analysing the latency performance of erasure codes. In *Information Theory (ISIT), 2014 IEEE International Symposium on*, pages 861–865. IEEE, 2014.

[311] Ali Shahbazi. *Computer Vision-Based Approaches to Neural Circuit Tracing at Scale.* University of Notre Dame, 2018.

[312] Ali Shahbazi, Jeffery Kinnison, Rafael Vescovi, Ming Du, Robert Hill, Maximilian Joesch, Marc Takeno, Hongkui Zeng, Nuno Maçarico Da Costa, Jaime Grutzendler, et al. Flexible learning-free segmentation and reconstruction of neural volumes. *Scientific reports*, 8(1):1–15, 2018.

[313] Nir Shavit. A multicore path to connectomics-on-demand. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 211–211. ACM, 2016.

[314] F.H. She, R.H. Chen, W.M. Gao, P.H. Hodgson, L.X. Kong, and H.Y. Hong. Improved 3d Thinning Algorithms for Skeleton Extraction. In *Digital Image Computing: Techniques and Applications, 2009. DICTA '09.*, pages 14–18, December 2009.

[315] Henry Shum and Leslie E Trotter Jr. Cardinality-restricted chains and antichains in partially ordered sets. *Discrete applied mathematics*, 65(1-3):421–439, 1996.

[316] HK Shum. Chains of bounded length and antichains of bounded width in partially ordered sets. 1990.

[317] Julian Shun. *Shared-Memory Parallelism Can be Simple, Fast, and Scalable.* Morgan & Claypool, 2017.

[318] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[319] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 152–163, New York, NY, USA, 2013. ACM.

[320] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM.

[321] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, pages 403–412, 2015.

[322] Parag Singla and Pedro Domingos. Entity resolution with Markov logic. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pages 572–582, Washington, DC, USA, 2006. IEEE Computer Society.

[323] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[324] Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

[325] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*, 33(4-6):889–906, 2018.

[326] Guy L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 218–231, New York, NY, USA, 1990. ACM.

[327] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *ICAC*, volume 13, pages 265–277, 2013.

[328] Josef Stoer, Roland Bulirsch, Richard H. Bartels, Walter Gautschi, and Christoph Witzgall. *Introduction to numerical analysis.* Texts in applied mathematics. Springer, New York, 2002.

[329] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[330] Márió Szegedy and Sundar Vishwanathan. Locality based graph coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 201–207, New York, NY, USA, 1993. ACM.

[331] Tolga Tasdizen, Pavel Koshevoy, Bradley C Grimm, James R Anderson, Bryan W Jones, Carl B Watt, Ross T Whitaker, and Robert E Marc. Automatic mosaicking and volume assembly for high-throughput serial-section transmission electron microscopy. *Journal of neuroscience methods*, 193(1):132–144, 2010.

[332] Fabian Tschopp. Efficient convolutional neural networks for pixelwise classification on heterogeneous hardware systems. *arXiv preprint arXiv:1509.03371*, 2015.

[333] Alan M Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[334] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *SPAA*, pages 83–94, 2016.

[335] Jacobo Valdes. *Parsing Flowcharts and Series-Parallel Graphs.* PhD thesis, Stanford University, December 1978. STAN-CS-78-682.

[336] Valgrind Developers. Massif: a heap profiler. Available at http://valgrind.org/docs/manual/ms-manual.html, 2018.

[337] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[338] Jacob Vogelstein. Machine intelligence from cortical networks (microns), 2016.

[339] Ashish Vulimiri, Oliver Michel, P Godfrey, and Scott Shenker. More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 13–18. ACM, 2012.

[340] Andrea Walther, Andreas Griewank, and Olaf Vogel. Adol-c: Automatic differentiation using operator overloading in c++. In *PAMM: Proceedings in Applied Mathematics and Mechanics*, volume 2, pages 41–44. Wiley Online Library, 2003.

[341] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 1967.

[342] R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8):463464, August 1964.

[343] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 314(1165):1–340, 1986.

[344] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[345] Wiki. Advanced vector extensions, 2016.

[346] WikiChip. Xeon platinum 8180 - intel, 2020.

[347] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 543–557, Oakland, CA, May 2015. USENIX Association.

[348] Hong Xu and Baochun Li. Repflow: Minimizing flow completion times with replicated flows in data centers. In *INFOCOM, 2014 Proceedings IEEE*, pages 1581–1589. IEEE, 2014.

[349] Jeonghee Yi, Farzin Maghoul, and Jan Pedersen. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *ACM International Conference on World Wide Web (WWW)*, pages 257–266, 2008.

[350] Hao Yin, Austin R Benson, Jure Leskovec, and David F Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 555–564, 2017.

[351] Adarsh Yoga and Santosh Nagarakatte. A fast causal profiler for task parallel programs. In *ESEC/FSE*, pages 15–26, 2017.

[352] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[353] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 325–336, 2009.

[354] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. *SIGARCH Comput. Archit. News*, 37(3):325–336, June 2009.

[355] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 712–721, New York, NY, USA, 1991. ACM.

[356] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[357] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[358] Jeremy Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 79, 2009.

[359] Aleksandar Zlateski, Kisuk Lee, and H Sebastian Seung. ZNN-A fast and scalable algorithm for training 3D convolutional networks on multi-core and many-core shared memory machines. *arXiv preprint arXiv:1510.06706*, 2015.