An Efficient Data Structure for Implementing Splitter Hyperobjects in Task-Parallel Systems

by

Qi Qi

S.B. Computer Science and Engineering, Mathematics
Massachusetts Institute of Technology (2021)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 26, 2021

Certified by .....
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

An Efficient Data Structure for Implementing Splitter Hyperobjects in Task-Parallel Systems

by

Qi Qi

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I present and analyze the novel stack-augmented split-tree data structure to support *splitter* hyperobjects for task-parallel systems. Splitters can mitigate races on shared, nonlocal state, where parallel nested tasks make independent local modifications without affecting shared history. The data structure is inspired by "persistent" trees, but refined to achieve optimal performance in the common case. I prove that in a program with n splitter variables using the mechanism based on the stack-augmented split-tree data structure, read and write accesses to a splitter variable cost $\Theta(1)$ except for the first access after a task migration, which costs $O(\log n + \log D)$ where D is the maximum depth of task nesting. This splitter data structure will enable the parallelization of search algorithms for computationally expensive applications, such as SAT solvers, theorem provers, and game-playing programs.

This thesis also contains theory and implementation work on other topics related to task-parallel programming and work stealing schedulers.

Some parts of this thesis represent joint work with William Kuszmaul.

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

Acknowledgments

Thanks to my advisor, Processor Charles Leiserson, for his wonderful guidance and advice throughout my MEng experience. Thanks to William Kuszmaul for all of his great insights during our technical discussions, as well as for his support and help on non-technical aspects of the thesis. Thanks to TB Schardl for his endless patience in answering all my questions about Cilk and computer systems. Thanks to Alex Iliopoulos for taking on the implementation side of the splitter project. Thanks to the rest of the Supertech research group for their support. Thanks to my friends and family for being really cool and keeping me sane.

This work was sponsored in part by the MIT-IBM Watson AI Lab and in part by by the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the author 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.

Contents

Chapter 1

Introduction

In an age where Moore's law is coming to an end, the importance of parallelization is increasingly clear. Individual processor cores are no longer doubling in speed every two years. As a result, increasing performance must come from other sources such as parallelization, which can speed up a program many-fold by dividing the necessary work among multiple cores.

Task-parallel programming, also known as dynamic multithreading or fork-join parallelism, is one model of concurrency. A programmer working in a task-parallel language can specify potential parallelism in a program without worrying about the details of how the instructions are mapped onto static threads and executed. Cilk, an extension to C and C++, is one example of a task-parallel language and the one that we consider in detail in this thesis. Opportunities for parallel work are created using the ***spawn*** keyword at the invocation of a function, which indicates that spawned function may execute concurrently with the ***continuation*** that follows the spawn [42]. Parallel loops can be decomposed efficiently by the compiler and the runtime system into nested spawns.

Parallelizing code is a difficult task, however, even with the intricacies of static-thread programming abstracted away. Concurrency introduces many potential sources of bugs, especially when it comes to variables or memory shared between threads. Nondeterministic behavior occurs when one thread updates a memory location while another thread reads the same location. This phenomenon, known as a determinacy

race [38], can cause programs to behave incorrectly.

Hyperobjects, introduced by Frigo et al. [41], make parallelization easier and help eliminate determinacy races. Using hyperobjects, a programmer can write parallel code that is similar to serial code, yet correct and safe from determinacy races. Existing ***reducer*** hyperobjects in Cilk, for instance, has seen much use. A reducer is defined by an associative binary reduction operation, and it is used to safely accumulate values together in parallel. Figure 1-1 demonstrates how an integer reducer with the addition operation can be used to compute the sum of an array of integers in parallel. Besides declaring the shared variable `x` as a reducer and adding loop parallelism with `cilk_for`, no other code needed to be modified to parallelize this program.

```
int sum(int* A, int N) {
  int x;
  for (int i = 0; i < N; ++i)
    x += A[i];
  return x;
}
```

```
int p_sum(int* A, int N) {
  int __reducer__((opadd)) x;
  cilk_for (int i = 0; i < N; ++i)
    x += A[i];
  return x;
}
```

Figure 1-1: A Cilk parallelization of a simple program using an integer sum reducer. `cilk_for` indicates that iterations of the loop may execute in parallel.

In addition to reducers, Frigo et al. proposed another natural kind of hyperobject called a ***splitter***. A splitter variable behaves as if at every spawn, the spawned function and the continuation each receive their local version of the variable. Any modifications made in the spawned function is not be seen in the continuation, and vice versa. The need for splitters arises naturally in algorithms that perform a search while modifying some non-local state, such as the simple SAT solver and game tree evaluators in chess-playing algorithms. Frigo et al. provided a sketch of a simple

splitter mechanism that does not achieve any strong theoretical performance guarantees. This thesis focuses on the design and analysis of the novel ***stack-augmented split-tree*** data structure for supporting splitters, which achieves provably good performance.

The desired splitter functionality can be attained by simply creating a copy of all splitter variables at every spawn, so that the spawned function and the continuation use different versions of each splitter. This solution is expensive, however, when the number of splitter variables is large, such as in a SAT solver where every variable in a large formula is a splitter. A full copy also seems wasteful when relatively few variables change in each spawn, as is the case for many search applications. The challenge lies in supporting the desired functionality for a large number of splitters while incurring low overhead.

A few approaches seem potentially applicable. The copy-on-write technique used in the OS `fork()` call, as described in [69, Chapter 10.3], seems to target this same problem. Every page is marked as copy-on-write upon a `fork()`, and a page is copied only when actually written to, reducing the cost of a `fork()` call. While performant enough in practice for the `fork` call use case, the copy-on-write technique does not give good theoretical guarantees — either the `fork()` call or the first write to each page has an asymptotically large cost.[1]

Another approach is to keep track of changes using lazily initialized maps. At a spawn, each of the two branched paths start with an empty map. An access to a splitter variable first attempts to look up the variable in the current map; if not found, it checks the maps stored at higher levels. This approach works well if there is only ever one spawn, but it does not work well when the forking structure of the program is complex. An access to a splitter may need to examine maps in a large number of ancestors before finding one that contains the target variable, resulting in potentially expensive accesses. The protocol proposed by Frigo et al. in [41] has similar issues.

[1] In a single-level page table covering n entries, either the page table size exceeds $\sqrt{n}$ and thus the cost of a `fork()` call exceeds $\Theta(\sqrt{n})$, or the page size exceeds $\sqrt{n}$ and thus the cost of the first write to each page exceeds $\Theta(\sqrt{n})$.

Another possible mechanism involves using persistent arrays. A fully persistent data structure keeps track of its change history in a "branching" model, so that every past version is available for queries and updates. Dietz's work in [31] shows that a fully persistent array of size n can support queries and stores for an amortized cost of $O(\log \log n)$ per operation, later proven to be asymptotically optimal in [70]. The state of n splitter variables in a task-parallel program can be represented by a fully persistent array of size n. Each splitter read and write access then simply performs a persistent query or store to the appropriate version of the fully persistent array.

This persistent-array mechanism suffers from three deficiencies, however. Firstly, Dietz's persistent array data structure is not naturally concurrent. If concurrent operations from different threads must be serialized using a lock to preserve correctness, then the running time of the program can be significantly impacted. Secondly, the $O(\log \log n)$ cost is amortized, and amortized analysis of a serial algorithm generally does not hold in the parallel setting. If a large number of expensive operations happen to fall onto one thread, then this program loses the guarantees that amortization provides, as these expensive operations cannot be amortized against cheap operations on the same thread. Lastly, a $O(\log \log n)$ overhead is not necessarily satisfactory. The stack-augmented split-tree mechanism presented in this thesis achieves a *constant* cost for read and write accesses in the "common case." A small number of accesses in the "uncommon case" are more expensive, but even those have a cost logarithmic in the number of splitters and the maximum depth of nested spawns.

To understand what constitutes the "common case," one must first understand how a parallel program is executed. This thesis assumes the use of the ***randomized work stealing scheduler***, which has provably good running time guarantees as analyzed in [1, 18] and as used in practice in Cilk. Upon startup, the scheduler creates a collection of threads called workers, typically one worker per processor in a multicore system. Each instruction in the program is subsequently mapped by the scheduler onto some worker for execution.

In some abstract sense, a spawned function and its continuation are equivalent: a spawn branches the computation into two valid paths, and either can be taken.

The scheduler does not treat the two equally, however. Upon encountering a spawn, a worker always executes the spawned function before the continuation. In the case that a worker runs out of work, it "steals" from another worker a continuation not currently being worked on and executes it.

The scheduler implements this functionality by having each worker maintain a deque, or double-ended queue, of function frames. When a function is spawned, its frame is pushed onto the bottom of the deque; when it returns, its frame is popped off the bottom. Thus, in the common case, each worker's deque operates like a call stack, and execution proceeds in the order that results from replacing all function spawns with serial function invocations. In the uncommon case where a worker runs out of work — that is, when the worker's deque becomes empty — the worker turns into a thief and steals the top frame from the deque of a randomly chosen victim worker. The thief then executes the continuation in the stolen frame. As analyzed in [18], steals are rare in a sufficiently parallel program.

The new stack-augmented split-tree mechanism supports n splitter variables in a task-parallel program and achieves the following guarantees:

1. The stack-augmented split-tree data structure supports full concurrency. It can even be implemented with no additional synchronization primitives (e.g. locks) whatsoever, as described in Appendix A.

2. Most read and write accesses incur $\Theta(1)$ cost. The first access to each splitter after a steal is the uncommon exception, and has $O(\log n + \log D)$ cost, where D is the maximum depth of nested spawns.

3. No garbage collection is required. Auxiliary memory allocations performed by the mechanism are freed under an explicit scheme in a safe and performant manner.

The purpose of this thesis is to lay down the theoretical groundwork for the stack-augmented split-tree mechanism. I describe in detail the data structure used, its supported operations, and how these operations tie into the work stealing scheduler

and the runtime. I also provide a detailed theoretical analysis proving properties of the mechanism. A preliminary implementation of splitter hyperobjects based on the stack-augmented split-tree mechanism has already been built by a different team.

The chapters on splitter hyperobjects are organized as follows. Chapter 2 provides background on task-parallel programming, the work stealing scheduler, and natural use cases motivating splitter hyperobjects. Chapter 3 provides a high level overview of the key technical points of the design of the mechanism and data structure. Chapter 4 presents a simple split-tree mechanism that does not offer the desired theoretical guarantees. Chapters 5 and 6 present the stack-augmented split-tree mechanism in full, building upon the simple mechanism. Chapter 7 proves the full mechanism's correctness. Lastly, Chapter 8 presents an analysis of the theoretical performance impact of using splitters following the stack-augmented split-tree mechanism.

This thesis also includes several other theoretical and practical results related to task-parallel progrmaming and work stealing schedulers. Chapter 9 investigates an alternative mechanism for reducer hyperobjects and evaluates its performance in practice. Chapter 10 presents a technique for bounding the concentration of random variables. It uses this technique to correct a long standing error in the analysis of contention in work stealing schedulers as presented in a seminal paper. Chapter 11 examines an intuitive parallel algorithm for the single-source shortest path problem and demonstrates that the parallel algorithm fails to achieve any speedup over its serial counterpart on a family of graphs.

Chapter 2

Background

This chapter provides background on task-parallel programming and work stealing. It also discusses motivation for splitter hyperobjects and describes the desired splitter functionality.

2.1 Task-Parallel Programming and the DAG Model

Task-parallel programming is a model of concurrency that allows the programmer to specify potential parallelism in a program, while abstracting away complexities of how the instructions are mapped onto static threads and executed. Cilk is one example of a platform that supports task-parallel programming. An overview of task-parallel programming can be found in [27, Chapter 27].

Parallel pseudocode under this model differs from regular, serial pseudocode by two concurrency keywords, ***spawn*** and ***sync***.

- The **spawn** keyword is used to create parallel work. Putting the **spawn** keyword before the invocation of a function indicates that the parent ***continuation*** — the code that immediately follows the spawn — may execute concurrently with the spawned child function.

- The **sync** keyword serves as a barrier, indicating that the function must wait for all of its spawned children to complete before proceeding to the statement after

sync. Every function implicitly syncs before it returns, preventing orphaned children.

An additional keyword, ***parallel***, can be used in conjunction with **for** to indicate that different iterations of a loop may all execute in parallel. Parallel loops are implemented using **spawn** and **sync** in a divide-and-conquer fashion.

For ease of later analysis, parallel programs must obey the following restrictions:

1. a **spawn** of a function is not immediately followed by a **sync**, and
2. there exists at least one un-synced **spawn** before a **sync**.

These restrictions are reasonable since these two constructs provide no useful parallelism. Identical behavior can be achieved by calling instead of spawning the function in the first case, and by removing the unnecessary sync in the second case. In fact, these optimizations are done automatically by the OpenCilk compiler.

One example of a task-parallel algorithm is the following simple (albeit extremely inefficient) parallel recursive algorithm for computing Fibonacci numbers. This algorithm executes recursive calls to FIB in parallel using **spawn**, and ensures that the results from these recursive calls are safe using **sync** before combining the results and returning.

```
1: function FIB(n)
2:     if n ≤ 1 then
3:         return n
4:     else
5:         x = spawn FIB(n − 1)
6:         y = FIB(n − 2)
7:         sync
8:         return x + y
9:     end if
10: end function
```

An execution of a task-parallel program can be modeled using an ***execution trace*** or trace, which is a directed acyclic graph (DAG). The trace of an execution can be defined and constructed in many ways. The approach taken in this work is similar to those used in [1, 18, 55], and quite different from the model used in [38].

The nodes of an execution trace consist of **spawns**, **syncs**, and ***strands***. Strands are sequences of instructions containing no parallel flow control (i.e. no **spawn**, **sync**, or return from **spawn**). A serial sequence of instructions may be broken into strands in many different ways, ranging from very coarse (one strand for the entire sequence) to very fine (one strand per machine instruction in the sequence). More frequently, we choose some in-between grain of division that is convenient for the situation. The ***cost*** of a node is defined to be the amount of time required to execute the instructions in the node. The cost of a **spawn** or **sync** node is considered to be unit time. In this theoretical model, we assume that the cost of a strand is fixed and not subject to variability due to caching and other system effects.

Directed edges between nodes of an execution trace represent the minimal dependencies between the execution order of strands. The existence of an edge from node A to node B indicates that instructions in node A must complete execution before instructions in node B are allowed to start. The set of edges is minimal in the sense that if a dependency between two nodes can be derived from existing edges, then an edge is not added between these two nodes.

With the exception of the start and end strands of a trace, a strand node has in-degree 1 and out-degree 1. A **spawn** node has in-degree 1 and out-degree 2, with one outgoing edge to the spawned child and one to the continuation. A **sync** node has in-degree at least 2 and out-degree 1, with one incoming edge for each spawned child, as well as one incoming edge from the strand corresponding to the continuation that comes before the **sync** in the program text.

It is useful to distinguish particular edges by type. The two out-edges of a **spawn** node are its ***spawn edge*** and its ***continuation edge***. The in-edge of a **sync** node from the parent continuation is its ***sync-continuation*** edge, and all other in-edges from spawned children are its ***sync-spawn*** edges. Edges which do not fall under the described categories are ***regular*** edges.

Figure 2-1 illustrates an execution trace for a call to FIB(3). This figure, as well as all illustrations of execution traces in the rest of this thesis, use the legend in Figure 2-2 for nodes and edges.

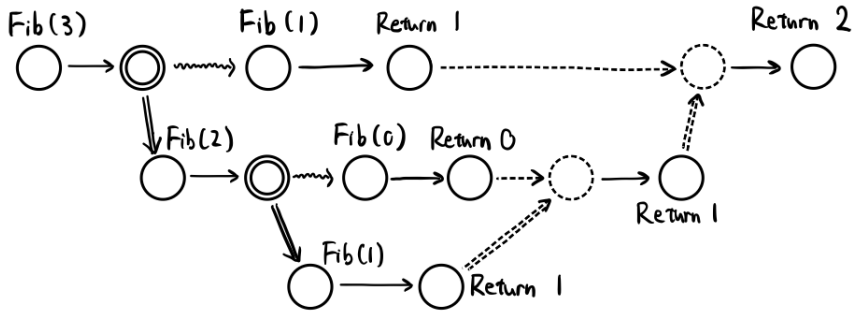

Figure 2-1: An execution trace for FIB(3).

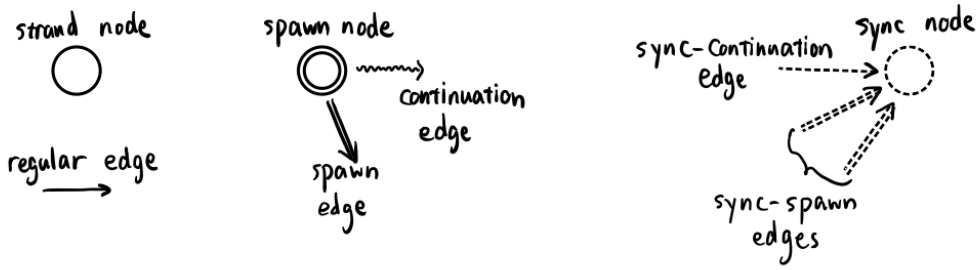

Figure 2-2: Legend for illustrations of execution traces.

The efficiency of a task-parallel algorithm can be gauged using two metrics: the work and the span. The ***work*** of an execution trace is the sum of the costs of all nodes in the trace. The ***span*** of an execution trace is the total cost of nodes on the most expensive path through the trace; this path is called the ***critical path***. The work of a trace can be interpreted as the amount of time required to perform the execution on a single processor, where all nodes must execution serially. The span can be interpreted as the amount of time required to perform the execution given an unlimited number of processors, since in such a situation, the running time is determined by the length of the longest chain of dependencies on instructions — in other words, the cost of the critical path. The running time of an execution trace on P processors is denoted by T_P. The work of a trace is thus expressed by T_1, and the span is expressed by T_∞.

2.2 The Cilk Runtime and Randomized Work Stealing Schedulers

The amount of time it takes to execute a task-parallel program depends not only on the structure of the execution trace and the cost of its nodes, but also on the way that the instructions in these nodes are mapped to processors for execution. This is the job of the ***scheduler***. This thesis assumes the use of the ***randomized work stealing scheduler***, such as the one used in the Cilk runtime.

At start up, the runtime system allocates a pool of threads called ***workers*** to cooperate in executing a task-parallel program. Each worker maintains a ***deque***, or double-ended queue, of ***call stacklets***. Each call stacklet consists of a number of function ***frames***, where the first frame in each call stacklet results from a **spawn**, and all other frames result from serial calls. Since we're only interested in what happens at parallel control points, we simplify the model by ignoring serial function calls. Worker deques are treated as containing spawned function frames.

When a function is spawned, its frame is pushed onto the bottom of its worker's deque. When it returns, its frame is popped off the bottom. Thus, in the common case, the worker's deque operates like a normal function stack in serial code.

When a worker runs out of work, it becomes a ***thief***. A thief performs a ***random steal*** by picking a ***victim*** uniformly at random among the other workers and attempting to remove the top frame from the victim's deque and place it on the thief's own deque. The random steal may fail if the victim's deque is empty or if other thieves concurrently target the same victim, in which case only one thief succeeds. The thief continues attempting random steals until it succeeds.

A frame becomes ***suspended*** if the worker that was executing this frame encounters a **sync** instruction that cannot be executed. A suspended frame is removed from the worker's deque, and the worker becomes a thief. Once all its spawned children complete, a suspended frame is resumed by the worker that executed the spawned child that returned last. This worker performs a ***provably good steal*** of the suspended parent frame, placing the frame onto its deque.

A more detailed description of Cilk's implementation of work stealing and runtime operations can be found in [41, 42].

Work-stealing achieves good running time as analyzed by Blumofe and Leiserson in [18]. Two key lemmas are restated here.

Lemma 1 (Lemma 12 in [18])**.** If a trace is executed on P processors using the randomized work stealing scheduler, then the expected total number of random steal attempts performed is bounded by $O(PT_\infty)$.

Lemma 2 (Lemma 13 in [18])**.** If a trace is executed on P processors using the randomized work stealing scheduler, then its expected execution time is bounded as

$$T_P \leq T_1/P + O(T_\infty).$$

2.3 Motivation and Desired Behavior of a Splitter Hyperobject

This section examines a class of algorithms with similar structure, with a particular focus on a naive algorithm for the boolean satisfiability problem. Parallelizing such algorithms naturally motivates the need for a runtime data structure like a splitter. The end of this section describes the desired functionality of a splitter in both intuitive and formal terms.

Algorithms with a DFS-like Structure

A large class of problems are naturally solved by algorithms that resemble depth-first search (DFS) in structure. Consider, for instance, the boolean satisfiability problem (SAT).

Problem 3. Given a boolean formula on a set of variables, find whether or not there exists some assignment of variables to boolean values that causes this formula to evaluate to true.

A simple, naive solution stores a partial assignment of booleans to variables in some global structure, representing the partial assignment currently under investigation. The algorithm recursively explores ways to complete the assignment, backtracking to reassign earlier variables when a partial assignment is shown to not lead to a valid solution. A pseudocode sketch for this algorithm is shown below.

Pseudocode for naive SAT solver

```
1: Variables x_0, ..., x_{n-1}
2: M mapping variables to {TRUE, FALSE, UNKNOWN}
3: function SOLVE(i)
4:     if i == n then
5:         return TRUE if formula satisfied by M, return FALSE otherwise
6:     else
7:         M[x_i] ← TRUE
8:         if SOLVE(i + 1) then return TRUE
9:         M[x_i] ← FALSE
10:        if SOLVE(i + 1) then return TRUE
11:        M[x_i] ← UNKNOWN
12:        return FALSE
13:    end if
14: end function
15: SOLVE(0)
```

This algorithm is intuitively straightforward to parallelize. Instead of serially checking the validity of assigning a variable to TRUE and the validity of assigning it to FALSE, the two paths can be explored in parallel. In the example pseudocode, one might imagine spawning off lines 9 and 10 to run in parallel with lines 7 and 8.

Unfortunately, this simple parallelization does not work due to the global variable M. Workers concurrently exploring different paths in the problem access and modify this same shared variable. Any modification to M performed by one worker is seen by all other workers exploring different parts of the space of possible assignments. These modifications from different workers conflict with each other, and the parallel algorithm fails to run correctly.

The naive SAT solver is not the only algorithm with this structure. For instance, some chess-playing algorithms compute the quality of a move by evaluating its game tree, which involves modifying a global representation of the chessboard by making

and unmaking a sequence of moves during traversal of the game tree's nodes. It is incorrect to directly parallelize such an algorithm by traversing the game tree in parallel, since modifications to the board representation at different points in the game tree interfere with each other. Sudoku solvers provide another such example.

One potential solution is to pass an immutable snapshot of the structure as an argument into each recursive call, instead of using a global, mutable structure. The solution based on this mechanism is expensive, however, if the number of variables is large, since it requires making a copy of this structure at every recursive call. Copying also seems wasteful, as most parts of this structure are unchanged at each recursive call — only a single boolean assignment is updated — yet the entire structure is copied.

The problem of parallelizing these algorithms can be solved efficiently by declaring these shared global variables as splitters.

Splitter Functionality

Intuitively, a ***splitter*** behaves like a variable whose value "splits" at every **spawn**, so that the spawned child and the parent continuation read and modify logically different copies of the splitter. Any modifications to the splitter value made by the spawned child cannot be seen by the parent continuation, and vice versa. When a spawned child returns, all modifications made in the spawn are discarded.

The following definition introduces a concept used to precisely state the functionality of a splitter.

Definition 4. Let v be a node of an execution trace G. The ***canonical path*** to node v is the unique directed path in G that starts at the start strand, ends at v, and does not pass through any **sync-spawn** edge.

The canonical path can be thought of as the path through the execution that does not enter any unnecessary spawns. The desired functionality can now be stated as follows.

Property 5. Let G be an execution trace, constructed so that every splitter read and write belongs to its own strand. Let v_{read} be a node in G performing a read to splitter X. Let v_{write} be the last node on the canonical path to v_{read} that performs a write to splitter X, if such a node exists. Then the value read in v_{read} equals the value written in v_{write}, or to the value at initialization if v_{write} does not exist.

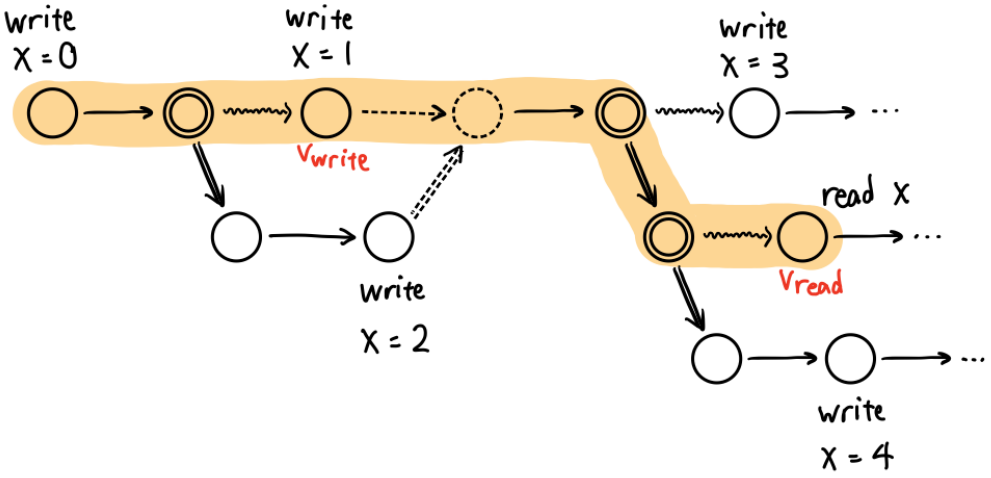

Figure 2-3: An execution trace snippet involving the splitter variable X. The canonical path to v_{read} is highlighted in orange. The value read at v_{read} is $X = 1$, since it is the value written in v_{write}, the last node performing a write to X on the canonical path.

Chapter 3

Technical Overview of the Stack-Augmented Split-Tree Mechanism

This chapter gives a high level summary of the key ideas behind the design of the data structure used to represent splitters. We start by exploring the simple split tree, which is a simplified variant of the data structure that serves as a starting point. We then move onto how the simple data structure can be augmented to achieve low theoretical overheads. We'll also gain some intuition on why the stack-augmented split-tree mechanism maintains correctness.

The Simple Split-Tree Mechanism

The simple split-tree data structure, described in detail in Chapter 4, is based on a fully persistent tree. The state of n splitters at each point in a computation is captured by the root node of a balanced binary tree with n leaves, as shown in Figure 3-1. The ith leaf of the tree stores the value of the ith splitter, and internal nodes store pointers to children but not to parents. Thus, there is some particular collection of n leaf nodes reachable from each root node, and the root node can be seen as a snapshot of the state of splitters taking on the values stored in these leaves.

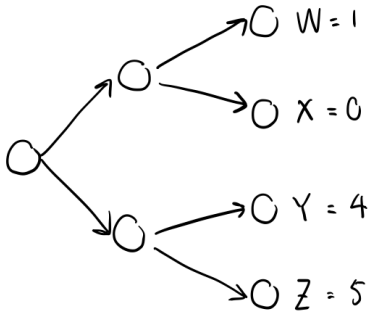

Figure 3-1: An example showing how a simple split tree represents the state of four splitters.

Writes are performed in a history-preserving manner, as nothing is ever modified. Instead, a write access creates a new version of the tree containing a mixture of new and old nodes. Writing to the ith splitter creates a new ith leaf node, as well as new versions of all nodes on the root-to-leaf path to the ith leaf. Other internal and leaf nodes remain unchanged. This operation called `path-copy` is illustrated in Figure 3-2.

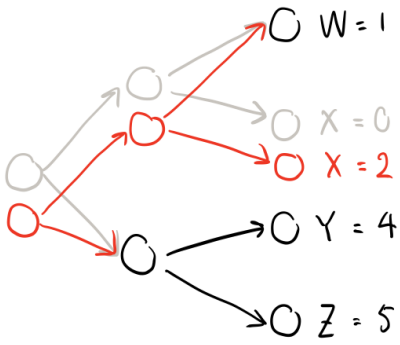

Figure 3-2: Updating splitter X calls `path-copy` on splitter X, creating new versions of all nodes on the root-to-leaf path to leaf X drawn in red.

In the mechanism based on this simple split tree, each frame keeps track of a split-tree root node. No special handling of the root node in the stolen frame takes place at a successful random steal. Writing to a splitter only updates the root node stored in the current frame. All other frames are unaffected, achieving the desired logical separation.

Read accesses under this simple split-tree mechanism must walk down the tree from a root node to the appropriate leaf node in order to access the value stored in the leaf. Write accesses must copy all nodes on a root-to-leaf path. Every read and write to splitter variables takes time $O(\log n)$, proportional to the height of the tree. This bound does not meet the goal of constant-time for most operations.

The Stack-Augmented Split-Tree Mechanism

The stack-augmented split-tree data structure, described in detail in Chapter 5, builds upon this simple variant by reducing the number of expensive root-to-leaf copies and root-to-leaf walks. Leaf nodes store not a single value, but a mutable array of values called a "leaf stack." In the common case, a write access to the ith splitter simply modifies the ith leaf stack. Only a small number of accesses result in a `path-copy`. As with the simple protocol, each frame keeps track of a split-tree root node.

Using mutable stacks may seem problematic. Different versions of the split tree stored in different frames often share nodes. It is now possible for a write access in one frame to make a change to a leaf stack which is visible from other frames. Maintaining logical separation is much less straightforward, relying on restrictions as to how these leaf stacks can be shared.

Each leaf stack is shared in a single-writer multiple-reader fashion, where only a serially executed section of the execution trace can modify any particular leaf stack. The trace can divided into a number of pieces called "chunks". Roughly speaking, a chunk starts when a worker performs a successful random steal and ends when this worker runs out of work on its deque and must steal again.[1] The "primary access", defined to be the very first access to a splitter variable in a chunk, results in a `path-copy` operation that creates a new leaf stack. This leaf stack may only be modified by subsequent write accesses to this splitter within this chunk.

After the primary access, the protocol is comparatively simple. The leaf stack created by the primary access is cached for lookup within the worker to avoid the cost of future root-to-leaf walks. Write accesses either append a new value to the leaf

[1] See Section 6.1.

stack or modify the last value stored in the leaf stack. Read operations return the last value in the leaf stack.

The primary access is more complex. When this access takes place, the leaf stack corresponding to the splitter contains values written at various points within some ancestor[2] chunk. This stack can even change under our feet — the chunk that created this leaf stack may still be ongoing and performing write accesses. The mechanism must extract the correct value from this stack. This task seems daunting. What additional information is associated with values written into these leaf stacks? What logic determines which of these values is correct to read at different points in the execution?

The key lies in the measure of "spawn depth," defined by the number of nested spawns at a given point in the execution. Each value in a leaf stack is associated with the spawn depth of the point in the program where the write took place. Write accesses maintain the invariant that entries in each leaf stack have distinct and strictly increasing spawn depths.

The primary access needs to extract the value corresponding to the appropriate spawn depth in the leaf stack created within some ancestor chunk. What is this appropriate spawn depth? Consider the point in the execution trace where the computation branches out from the ancestor chunk due to a steal, and where the branch eventually leads to this access. Write accesses in the ancestor chunk before this branching point should be reflected in the value read, while write accesses after this branching point are logically separate and should not be read. It is thus correct to read the value associated with the largest spawn depth that does not exceed the spawn depth of this branching point.

It remains to solve the problem of tracking the spawn depths of these branching points. The leaves of different splitters may have been created inside different ancestor chunks, and thus different splitters may have different branching points. A straw-man approach iterates through every splitter when a steal occurs. For each splitter that had a new leaf created, its branching point spawn depth is updated to the spawn depth

[2]See Definition 9.

of this steal. This approach has the clear downside that steals take an asymptotically large amount of time, which breaks the fundamental assumption in the work stealing scheduler that steals take constant time.

The stack-augmented split-tree mechanism performs the necessary updates at a steal in constant time. The branching point spawn depths of splitters returned by the operation `depth-query` are tracked through values written on the edges of the split tree in a cascading manner. Calling `depth-query` on a splitter returns the last non-NIL value (if any) encountered on the root-to-leaf walk. When a new leaf is created, the `path-copy` operation updates the values on edges appropriately so that all edges on the path to the new leaf have a value of NIL, while `depth-query` on every other leaf is unchanged.[3] At a steal, the value written on the top-level edge is updated to the spawn depth of the steal, which updates `depth-query` on exactly those splitters with new leaves.[4]

Figure 3-3 illustrates an example of splitters represented using the stack-augmented split tree.

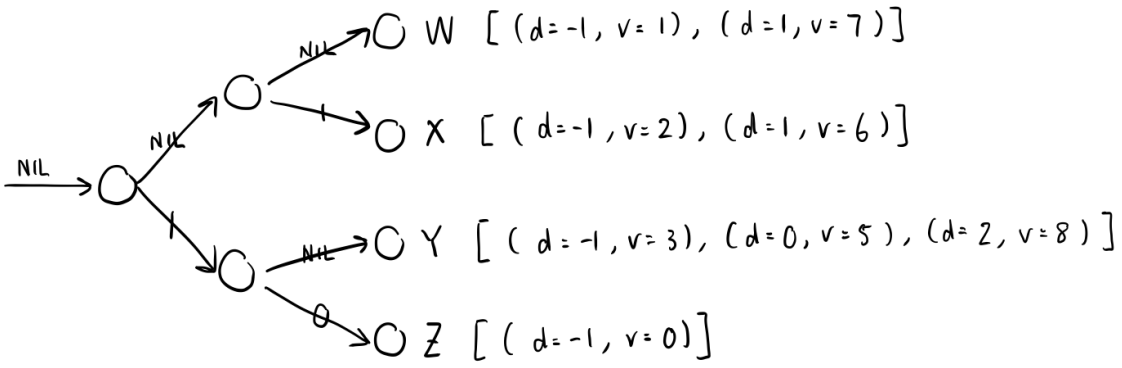

Figure 3-3: An example shwoing how the stack-augmented split-tree represents four splitters.

Additional care must be taken to ensure that the stack-augmented split-tree mechanism cleans up after itself properly and frees all memory allocated for nodes created at `path-copy` operations. Responsibility for allocations is determined in two stages

[3] See Property 6.
[4] See Property 7.

based on chunks. Chapter 6 describes how memory can be explicitly freed in a safe and thorough manner.

Chapter 4

Simple Split-Tree Mechanism

This chapter presents a simplified variant of the mechanism called the ***simple split-tree*** mechanism. The simple split-tree mechanism does not achieve the desired theoretical overhead, as no access has a cost of $\Theta(1)$. Nonetheless, the data structure used in the simple split-tree mechanism serves as a foundation for that used in the stack-augmented split-tree mechanism. Understanding this simplified mechanism aids the understanding of the complexities of the augmented mechanism.

Section 4.1 presents the simple split-tree mechanism assuming that memory allocations are freed through automatic garbage collection. Section 4.2 removes this assumption and examines explicitly memory management, necessary in a language like Cilk. This structure mirrors the presentation of the augmented mechanism in Chapters 5 and 6.

4.1 Garbage-Collected Simple Split-Tree Mechanism

This section temporarily assumes that memory allocations performed by the mechanism are freed through automatic garbage collection.

We'll start by learning about the simple split-tree data structure and the operations it supports. Then, we'll look at the actions of the runtime system at each

runtime operation. Lastly, we'll quickly analyze the theoretical performance overhead of this mechanism, finding that every splitter access costs $O(\log n)$ time in a program that uses n splitters.

Data Structure

Simple split tree

The state of n splitters at each point in time is represented by a simple split tree: a balanced binary tree with n leaves, where each leaf corresponds to a splitter and stores a value.[1] Each internal node of the tree contains pointers or edges to its child nodes. It notably does not contain pointers to any parent node.

A split tree is accessed through the use of a ***handle***, which is a pointer (represented as a directed edge) to the root node of the tree. The handle is said to ***attach*** to the split tree whose root it points to. All parts of this tree can be accessed by following parent-to-child edges/pointers starting at the root.

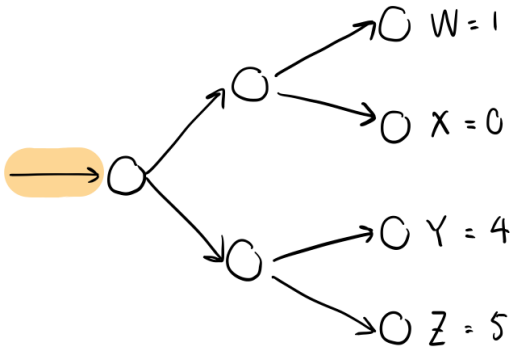

Figure 4-1: An example of a handle (highlighted in orange) leading to a split tree in a parallel program using four splitter variables.

Simple split trees support the following operation:

`path-copy(handle, splitter, new-val)`

[1] "Balanced" means that the depth of every leaf is $O(\log n)$. While that is the only requirement in order for the theoretical analysis to hold, diagrams will picture split trees that are full and complete.

This operation takes as arguments a handle, a splitter variable, and a new value for the splitter. It updates the handle to attach to a new tree. The new tree shares many of its nodes with the tree previously attached to the handle, but all nodes lying on the path from the leaf of the input splitter to the root are new. No part of the old tree is modified. This operation is illustrated in Figure 4-2.

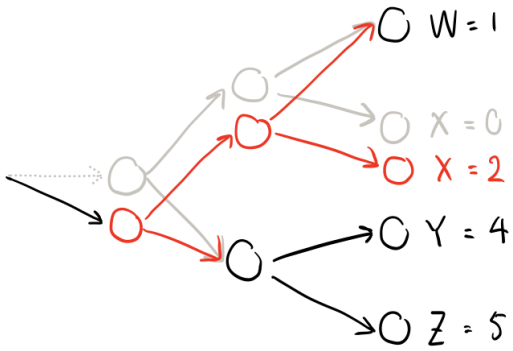

Figure 4-2: `path-copy` is performed on splitter X, updating it to a new value of 2. This process creates new nodes in red. The new tree is drawn in red and black, and the old tree is drawn in grey and black. No part of the old tree is modified by the `path-copy` operation. The handle is updated from pointing to the root in grey to the new root in red.

The `path-copy` operation modifies splitters in a history-preserving way. The split tree accessible from each root represents the state of splitters at some point in history.

In a split tree with n leaves, `path-copy` takes $O(\log n)$ time to run, as it walks down $O(\log n)$ nodes in the tree and creates $O(\log n)$ new nodes.

Protocol

State Maintained

To support splitters, every spawned function frame keeps track of a handle.

The spawned frame at the bottom of each worker's deque — the frame closest to currently executing code — is important for describing the splitter protocol. For ease of explanation, the handle stored in this spawned frame is called the ***worker handle***, as it is the handle most actively operated on by the worker.

Operations

Read access

Start from the worker handle and traverse to the leaf node of the target splitter. Return the value stored in the leaf node.

Write access

Perform `path-copy`, using as arguments the worker handle, the target splitter, and the value to be written.[2]

Spawn

Set the handle in the newly spawned frame to be a copy of the previous worker handle (the handle in the previously bottommost spawned frame).

Return from spawn

No-op. The previous worker handle is discarded and any writes performed in this spawn lost, as is expected.

Random steal

No-op. The stolen frame contains the worker handle to be used by the thief going forward.

Sync

No-op. If the sync fails, the frame containing the worker handle becomes suspended, and later correctly reinstated.

Provably good steal of parent

No-op. The suspended spawned frame reinstated on the worker deque contains the worker handle to be used going forward.

[2]This can result in unnecessary `path-copy` operations. For instance, if there is no parallel control flow between two writes to the same splitter, it is safe for the second write to simply modify the value stored in the leaf node created in the first write. We won't worry about optimizing for this case in the simple mechanism.

Theoretical Overhead

The theoretical performance overhead of the simple split-tree mechanism is straightforward to analyze. A write access must perform a `path-copy` operation, which requires creating a root-to-leaf path of length $O(\log n)$. A read access must traverse a root-to-leaf path of length $O(\log n)$. Therefore, every splitter read and write access takes time $O(\log n)$.

Other runtime operations are unaffected. Spawns, returns, and random steals still take constant time.

4.2 Explicit Memory Management in the Simple Split-Tree Mechanism

This section examines how memory allocations performed by the mechanism can be explicitly managed and freed without relying on garbage collection. We'll discover that a simple memory management scheme can have a surprisingly high impact on performance. This phenomenon will also later be observed to a lesser extent in the stack-augmented mechanism.

Protocol

State Maintained

Every spawned function frame keeps track of a log, which is a linked list of pointers to memory allocations performed by the split tree. The log stored in the bottommost spawned frame on each worker's deque is called the ***worker log***.

The memory needed for new nodes created in a `path-copy` operation can be allocated in a single piece. Each `path-copy` operation therefore produces a single memory pointer to add to a log.

Operations

Write access

Add a pointer to the memory allocation performed by `path-copy` to the worker log.

Spawn

Set the log of the new spawned frame to be an empty linked list.

Return from spawn

Free all memory allocations those pointers are stored in the worker log, taking time proportional to the size of the log.

All other operations

No-op.

Theoretical Overhead

The memory deallocations performed at return-from-spawns can have a surprisingly high impact on the parallelism of an execution trace.

The number of frees done at a return-from-spawn equals the number of splitter write accesses performed within the body of the spawn. One may mistakenly reason that the cost of these frees can be amortized against the cost of splitter writes, and therefore have no more than a constant factor impact on the work and span of an execution trace. This logic is incorrect regarding the span, however, as demonstrated in the following example.

When we ignore the cost of deallocations performed at return-from-spawns, the span of this program's execution is $\Theta(M)$. The longest path through an execution of this program passes through M spawns, M writes to splitters, and M returns. Each spawn and splitter write takes constant time and, ignoring deallocations, each return-from-spawn takes constant time.

Example where memory frees at return-from-spawns heavily impacts parallelism

```
1: Splitter X
2: Int M
3: function EXAMPLE(m)
4:     if m > 0 then
5:         spawn EXAMPLE(m-1)
6:     end if
7:     for i = 1 ... M do
8:         Write X ← i
9:     end for
10:    sync
11: end function
12: function MAIN
13:    EXAMPLE(M)
14: end function
```

Taking the cost of deallocations into account, every return-from-spawn takes M time as M deallocations need to be performed. The previously described path through the execution now has a length of $\Theta(M^2)$, which is equal to the work of the execution. The parallelism of the program has dropped from $\Theta(M)$ to $\Theta(1)$.

We will see similar issues in the stack-augmented split-tree protocol regarding costs at return-from-spawns that cannot simply be amortized away.

Chapter 5

Garbage-Collected Stack-Augmented Split-Tree Mechanism

This chapter describes the stack-augmented split-tree mechanism, temporarily assuming that memory allocations performed by the mechanism are freed automatically through garbage collection. Making explicit memory management efficient and safe is a complex problem examined in detail in Chapter 6.

The augmented mechanism builds on the ideas from the simple mechanism. It uses a more sophisticated version of the split tree that minimizes the number of expensive `path-copy` operations required.

One key definition used throughout the protocol is the ***spawn depth***. The spawn depth of a point in an execution trace is the depth of nested spawns. For example, the spawn depth of every instructions in the trace of a serial program is 0.

Section 5.1 describes data structures used in the mechanism and their supported operations. Section 5.2 specifies how these data structures are used in the runtime system and what happens at various runtime operations. Section 5.3 examines in detail several example code snippets that use splitters, in order to illustrate subtleties in the mechanism.

5.1 Data Structures

This section describes key data structures used in the protocol and the operations that they support.

Dynamic array

The dynamic array supports `create`, `append`, `pop`, `get` at any index, `modify` at any index, and `destruct`, all in $O(1)$ time. These costs are *not amortized*.

A dynamic array uses a reader-writer lock, so that all `append` and `modify` operations are serialized and do not happen concurrently with `get` operations.[1] When all writes take place sequentially, the use of a reader-writer lock does not impact performance by more than a constant factor.

The dynamic array is used extensively throughout the splitter protocol. Deamortizing the cost of operations is key in its design, as data structures whose operations have amortized costs impact the span of an executions in unpredictable ways.

A dynamic array keeps track of pointers to two pieces of memory, the `primary` and `backup`, as well as two integers `p-ind` and `b-ind`, which track the number of items in `primary` and `backup` respectively.

Operations are performed as follows:

Create

Set `primary` to be a small, newly allocated array (e.g. size 2). Set `backup` to a newly allocated array double the size of the `primary`. Set `p-ind` and `b-ind` to 0.

Append

Place the new item in the next (`p-ind`th) slot in the `primary`, increment `p-ind`. Copy up to two items from the `primary` to the `backup`. To be precise, first set the `b-ind`th item in the `backup` to be equal to the `b-ind`th item in the

[1] It is possible to use dynamic arrays without any locking mechanism, which may be of interest for particular theoretical models of computation. Appendix A describes changes to the dynamic array data structure and to the splitter mechanism that eliminate the need for locks.

`primary`, and increment `b-ind`. If `b-ind` is still less than `p-ind`, meaning that the `backup` is lagging behind the primary, then copy the next element as well and increment `b-ind` again.

If the primary is now full after the new append, call `resize-up`.

Resize-up

Rearrange the internal structure to double the size of both `primary` and `backup` as follows:

Free the memory of `primary`.

Set the new `primary` to point to the old `backup`.

Allocate a new piece of memory that is double the size of the old `backup`, point `backup` to the new memory, and set `b-ind` to 0.

Pop

Decrement `p-ind`. If `b-ind` is now larger than `p-ind`, decrement `b-ind`.

If `primary` is now no more than 1/4 filled, and it is larger than its initial size, call `resize-down`.

Resize-down

Free the second half of both `primary` and `backup`.

Get

Return the item at the desired index in `primary`. If the desired index is larger than `p-ind`, return that the read is invalid.

Modify

Modify the value at the desired index in `primary`. If `backup` is long enough that it includes the desired index, also modify the value at the desired index in `backup`.

Destruct

Free both `primary` and `backup`.

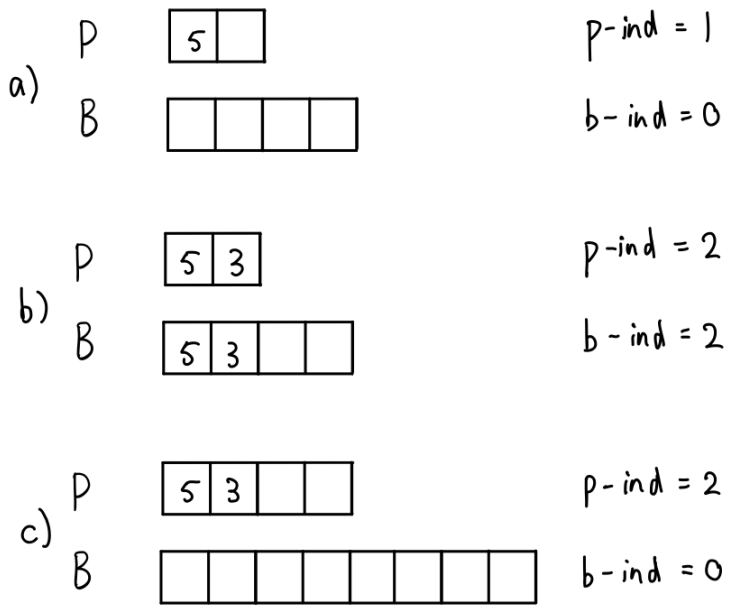

Figure 5-1: An example of calling `append` of value 3 on a dynamic array, which then triggers `resize-up`.

The only question regarding correctness is whether any data is lost during a `resize-up`, or, in other words, whether `backup` contains all data in `primary` when `resize-up` is called. This can be proven as follows:

- Consider each `resize-up` operation R in a sequence of operations on a dynamic array. Let S be the latest `create`, `resize-up`, or `resize-down` operation that occurred before R, so that the only operations between S and R are `append`s and `pop`s.

- Suppose that after S, the `primary` has a capacity of $2x$. By construction, right after S, at most x items are filled in `primary`. Therefore, the difference between `p-ind` and `b-ind` is at most x right after S.

- Since the `primary` must be full in order for the `resize-up` operation R to be called, at least x `append` operations must take place between S and R.

- Whenever the difference between `p-ind` and `b-ind` is nonzero, an `append` or

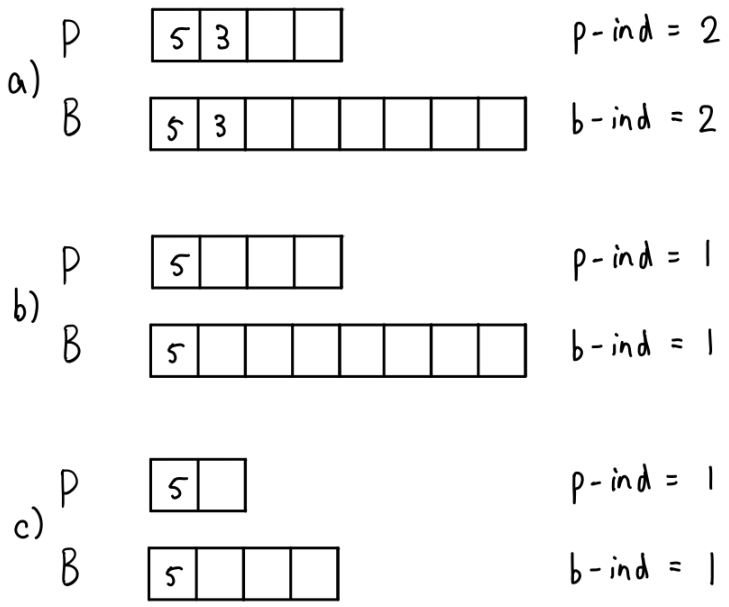

Figure 5-2: An example of calling `pop` on a dynamic array, which then triggers `resize-down`.

`pop` operation decreases this difference by 1. Whenever the difference is zero, an `append` or `pop` maintains the difference to be zero.

- By the time the `resize-up` operation R is called, the difference between `p-ind` and `b-ind` has decreased to zero as at least x `append` operations have taken place. Thus, `backup` contains all data in `primary` when `resize-up` is called.

All diagrams below only illustrate items contained within each dynamic array. The details of internal working of dynamic arrays are abstracted away.

Stack-augmented split tree

Similar to in the simple mechanism, the state of n splitters is represented by a split tree: a balanced binary tree with n leaves. Unlike the simple split tree, the stack-augmented split tree contains additional information as follows:

- Each child pointer (parent-to-child edge) is associated with a value `depth`, which

is either NIL or a non-negative integer.

- Each leaf node contains not a single value, but a dynamic array of `(depth, value)` pairs. Such an array is called a ***leaf stack***.

As before, a split tree is accessed using a handle, which is a pointer/edge to the root node of the tree. Much like edges in the stack-augmented split tree, the handle to a stack-augmented split tree is associated with a value `depth`. The handle is said to attach to the split tree whose root it points to.

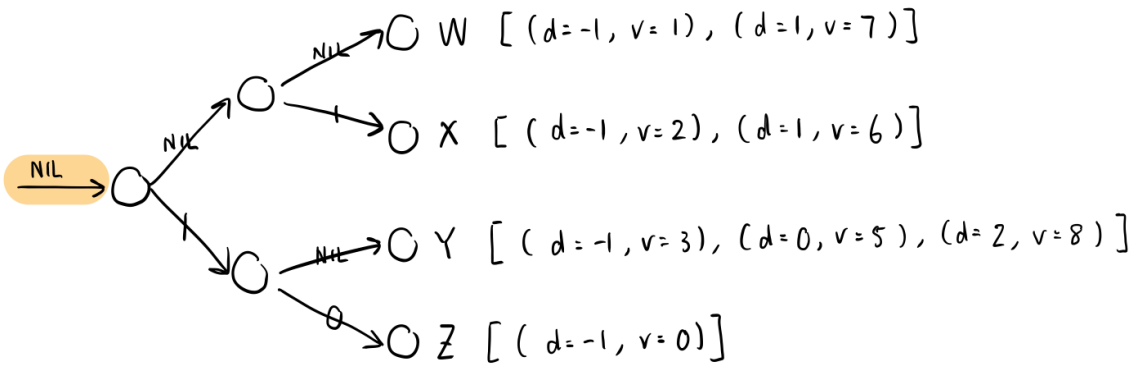

Figure 5-3: An example of a handle (highlighted in orange) leading to a stack-augmented split tree in a program using four splitter variables.

Stack-augmented split trees support the following operations:

`depth-query(handle, node)`

This operation takes as arguments a handle, and any node in the split tree that the handle is attached to. It returns either NIL or a non-negative integer. This operation walks down the tree, traversing a sequence of edges starting with the handle and ending with the edge pointing to the target node. If all edges on this path have a NIL depth, then the operation returns NIL. Otherwise, the operation returns the *last* non-NIL depth encountered along this walk.

As a shorthand, `depth-query(handle, splitter)` is generally written in place of `depth-query(handle, leaf node corresponding to splitter)`.

For example, for the split tree illustrated in Figure 5-3,

- `depth-query(handle, W)` returns NIL,
- `depth-query(handle, X)` returns 1,
- `depth-query(handle, Y)` returns 1, and
- `depth-query(handle, Z)` returns 0.

`path-copy(handle, splitter, new-val)`

This operation takes as arguments a handle, a splitter variable, and a new value for the splitter. It updates the handle to attach to a new split tree without modifying any part of the old tree, as follows:

Let the root-to-leaf path in old (input) split tree consist of nodes $x_0, ..., x_k$, where x_0 is the root and x_k is the leaf node corresponding to the input `splitter`. Denote by c_{i+1} the child of x_i other than x_{i+1}. Construct a new root-to-leaf path, $x'_0, ..., x'_k$, as follows:

- Set the child pointers/edges of x'_i to point to x'_{x+1} and c_{i+1}.
- Set depth of edge $x'_i x'_{i+1}$ to NIL.
- Set depth of edge $x'_i c_{i+1}$ to `depth-query(`x_0`,` c_{i+1}`)`.
- Create a new leaf stack for node x'_k that only contains the `(depth, value)` pair `(-1, new-val)`.

Finally, update the handle to a depth of NIL and point it to the new root x'_0.

In a stack-augmented split tree with n leaves, `path-copy` takes $O(\log n)$ time to run. Note that `depth-query(handle,` c_{i+1}`)` for all $O(\log n)$ values of i can be computed in a total of $O(\log n)$ time as follows:

- Compute the sequence of values `depth-query(handle,` x_i`)` iteratively in $O(\log n)$ time, by computing each `depth-query(handle,` x_{i+1}`)` using the previous term of `depth-query(handle,` x_i`)` and the depth of edge $x_i x_{i+1}$.
- Compute each `depth-query(handle,` c_{i+1}`)` using `depth-query(handle,` x_i`)` and the depth of edge $x_i c_{i+1}$.

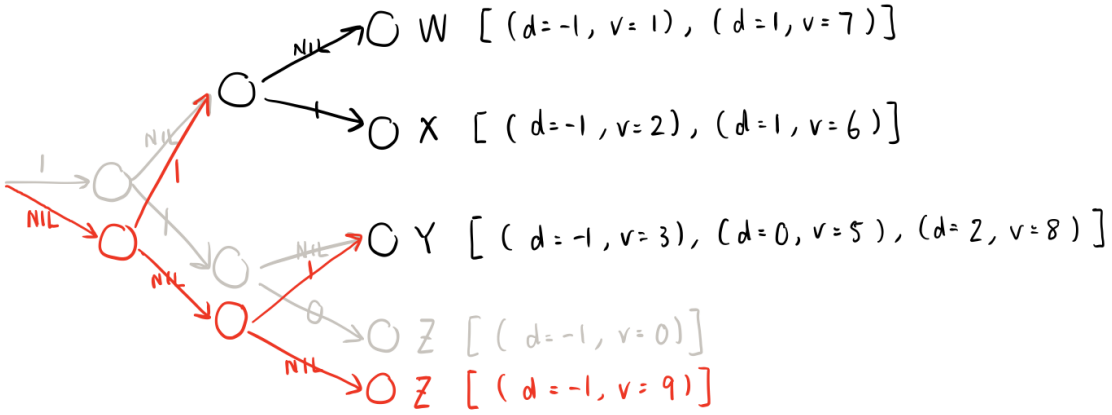

Figure 5-4: `path-copy` is performed on splitter Z with a new value of 9. This process creates new nodes and dynamic array in red. No part of the old split tree, drawn in grey and black, is modified. The handle is updated from pointing to the old root in grey and having a depth of 1, to pointing to the new root in red and having a depth of NIL.

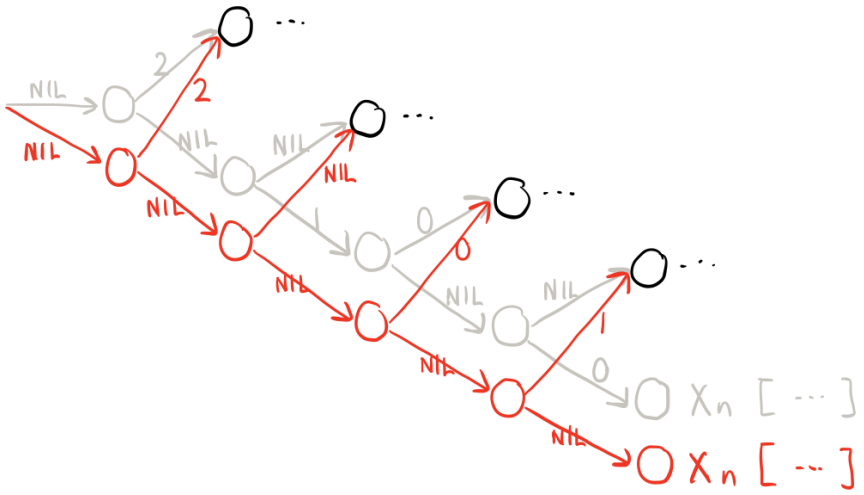

Figure 5-5: `path-copy` is performed on splitter X_n in a larger split tree. Much of the tree is unaffected by this operation and omitted from the drawing.

The `path-copy` operation guarantees the following key property, evident by construction.

Property 6. Let `path-copy(handle, input-splitter, new-val)` update the contents of the handle from `old-handle` to `new-handle`. Then

- `depth-query(new-handle, input-splitter)` equals NIL, and
- for all other splitters `s` distinct from `input-splitter`,

  `depth-query(new-handle, s)` equals `depth-query(old-handle, s)`.

`handle-update(handle, new-depth)`

This operation takes as arguments a handle and an integer depth value. If the depth of the input handle is NIL, it updates the depth to `new-depth`. Otherwise, it makes no modification.

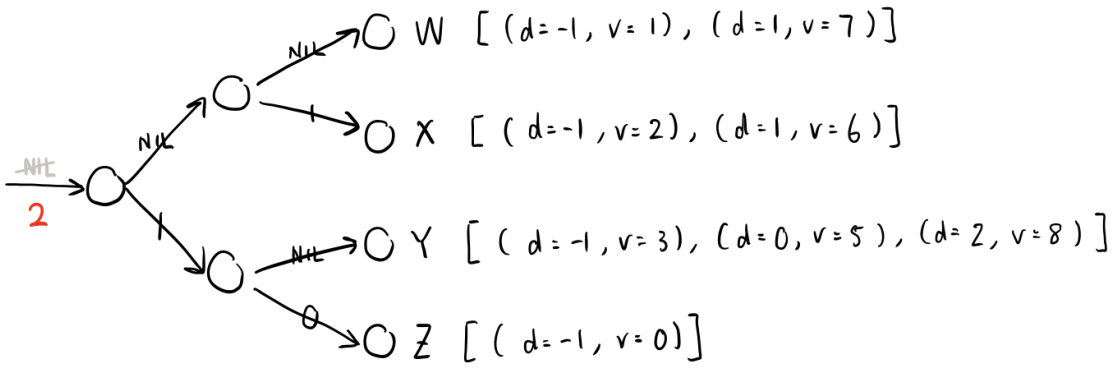

Figure 5-6: `handle-update` is performed with a depth of 2, updating the depth of the handle from NIL, written in grey, to 2.

The `handle-update` operation guarantees the following key property, evident by construction.

Property 7. Let `handle-update(handle, new-depth)` update the contents of the handle from `old-handle` to `new-handle`. Then for each splitter `s`,

- if `depth-query(old-handle, s)` equals NIL, then

  `depth-query(new-handle, s)` equals `new-depth`, and
- if `depth-query(old-handle, s)` does not equal NIL, then

  `depth-query(new-handle, s)` equals `depth-query(old-handle, s)`.

Cache

A cache is a map taking splitter variables to leaf stacks. The cache supports constant time reads and writes.

Record

A record is a collection of values. It supports constant time `create`, `insert`, and `destruct`, as well as linear-work log-span parallel traversal of its contents. The underlying data structure of a record can be, for instance, a dynamic array.

5.2 Protocol

The protocol operates very differently at the first access to a splitter after a steal, called the ***primary access***, and at all other accesses, called ***secondary accesses***. At a primary access, the protocol searches an existing leaf stack for the correct value to read, then performs a `path-copy` operation to create a new leaf. Information about what values must be found in these searches is tracked through depth values on edges of the split tree. At a secondary access, the protocol performs a constant time operation on a cached leaf stack.

State Maintained

Every spawned function frame keeps track of a handle and a record of splitter identities. Every worker keeps track of a cache. In addition, suspended frames temporarily hold a cache.

The handle and record of the bottommost spawned frame on each worker's deque are called the ***worker handle*** and ***worker record***, as they are most actively operated on by the worker throughput the protocol.

The record is used to track which splitter leaf stacks have been appended to.

Initialization

At startup of the runtime system, one worker has the *Main* frame as the only frame on its deque. The initial handle contained in the *Main* frame has the following properties:

- The handle is associated with a depth of NIL.
- Every edge in the split tree that the handle attaches to is associated with a depth of NIL.
- Every leaf stack contains a single entry of `(-1, init-val)` where `init-val` is the initial/default value of each splitter.

The worker starts with a fully populated cache, mapping each splitter to the corresponding leaf stack in the initial tree.

Operations

Read access

If the splitter is present in the cache, return the value stored in the last entry of the cached leaf stack.

If the splitter is not present in the cache, start from the worker handle and traverse to the leaf stack `leaf` of the target splitter. Compute `depth-query(worker handle, splitter) = d-leaf`, which is guaranteed to be non-NIL. Binary search[2] `leaf` to find the pair `(d-found, v-found)` such that `d-found` is the largest depth less than or equal to `d-leaf`. Perform `path-copy(worker handle, splitter, v-found)`. Cache the newly created leaf stack. Return `v-found`.[3]

Write access

[2]One needs be careful with invalid values seen in the binary search, as the array being searched may shrink during the search due to another worker performing `pop` operations. An invalid value (where the index being read is greater than the current length of the array) must be treated as reading a too-large depth.

[3]This protocol performs some unnecessary `path-copy` operations. One can optimize the protocol to only perform `path-copy` on write accesses instead of on read accesses. This optimization may be important in practice, but as it does not impact the theoretical overhead, we won't worry it in this work.

If the worker cache does not contain this splitter, first perform a read access to this splitter.

Let `d-spawn` be the current spawn depth. Examine the last `(d, v)` pair in the cached leaf stack. If `d` equals `d-spawn`, then overwrite `v` with the new value. Otherwise, append `(d-spawn, new value)` to the cached leaf stack, and add this splitter to the worker record.

Spawn

Set the handle in the newly spawned frame to be a copy of the previous worker handle (the handle in the previously bottommost spawned frame).

Set the record in the newly spawned frame to be empty.

Return from spawn

Update the handle in the newly bottommost spawned frame (the new worker handle) to the value of the handle in the just-returned spawned frame (the old worker handle).

Iterate over the record in the just-returned spawned frame (the old worker record). For each iterated splitter, perform `pop` on its cached leaf stack.

Random steal

Let `d` be the current spawn depth (at the stolen continuation) and let `handle` be the handle in the stolen frame. Perform `handle-update(handle, d)`.

Destroy the record in the stolen frame. Create a new empty record and set it to be the new worker record.

Invalidate/clear the worker cache.

Sync

If the sync fails and the frame becomes suspended, store the worker cache in the suspended frame. The cache will later be correctly reinstated.

Provably good steal of parent

Destroy the current worker cache and reinstate the cache stored in the previously suspended frame as the worker cache.

5.3 Examples

In this section, we examine several example program snippets to highlight subtleties in the mechanism. The goal is to ensure understanding of the mechanism and to provide some intuition for the upcoming proofs of correctness and performance.

Each example is presented as a sequence of chronological events in an execution of a parallel program. Workers are referred to as W_i. Splitters are referred to as S_i, and every splitter is initialized to have a value of `init-val`. All example programs in this section use four splitter variables.

Victim reads correct value after steal

Steps:

1. W_1 enters spawned f.
2. W_2 steals the $Main$ frame from W_1.
3. W_2 writes `new-val` to S_4.
4. W_1 reads S_4.

The state of the system after each step in this example is illustrated in Figure 5-7.

In step 2, thief W_2 steals a continuation from victim W_1. The mechanism must ensure that even though the thief modifies the value of a splitter in step 3, the victim is unaffected and reads the correct, unmodified value of this splitter in step 4.

This is satisfied as W_2 performs `path-copy` on its primary access to S_4 in step 4, creating a new leaf stack without modifying the previous leaf stack. The updated value of `new-val` only appears in the newly created leaf stack drawn in red. The read access in step 4 examines the old leaf stack reachable from the worker handle of W_1, which only contains `init-val`, and correctly return the value in the last entry of this array.

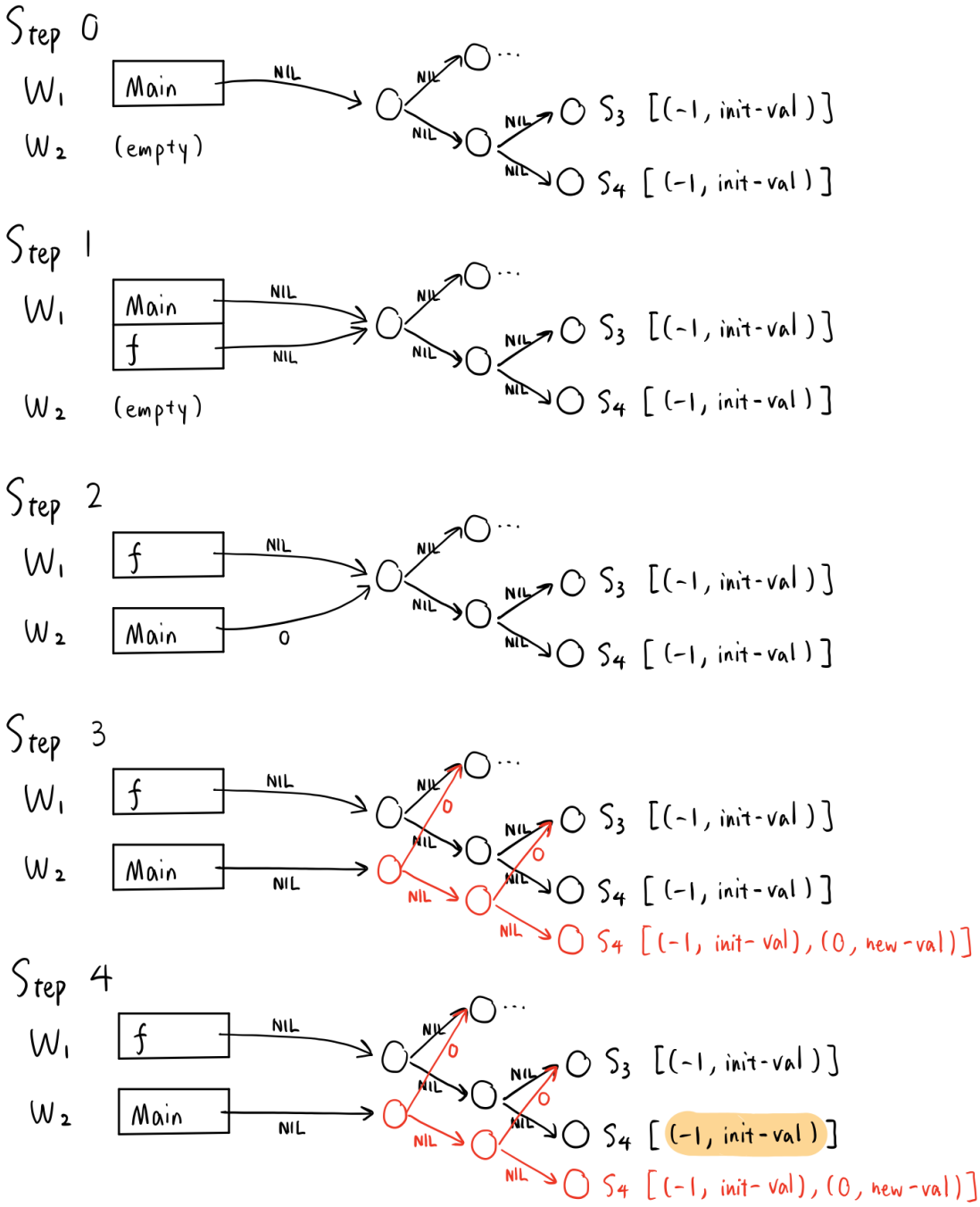

Figure 5-7: Boxes on the left represent worker deques of spawn frames. The edge leading out of each frame represents the handle stored in this frame. Nodes present since the start of the program are drawn in black, and nodes created by `path-copy` in step 3 are drawn in red. The entry containing the accessed value in step 4 is highlighted in orange.

Thief reads correct value after steal

Steps:

1. W_1 writes `val-1` to S_4.
2. W_1 enters spawned f.
3. W_2 steals the $Main$ frame from W_1.
4. W_1 writes `val-2` to S_4.
5. W_2 reads S_4.

The state of the system after each step in this example is illustrated in Figure 5-8.

In step 3, thief W_2 steals a continuation from victim W_1. The protocol must ensure that even though the victim modifies the value of a splitter in step 4, the thief is unaffected and reads the correct value in step 5.

The thief W_2 performs its primary access to splitter S_4 in step 5, which involves searching through the old leaf stack to find the entry with the appropriate associated depth. W_2 starts by calling `depth-query` on S_4 to find the target depth in the search; this operation walks along the path highlighted in green and returns a depth of 0. W_2 then searches the leaf stack of S_4 to find the entry with the largest depth less than or equal to the target of 0, finding the entry `(0, val-1)` highlighted in orange. The write of value `val-2` to splitter S_4 performed by worker W_1 in step 4 does not affect the result of this search, since the write happened at a spawn depth of 1, larger than the target depth of the search.

After the entry is found, W_2 performs `path-copy` to splitter S_4. The value in the found entry is used to populate the newly created leaf stack. Lastly, the value in the found entry is returned.

Worker keeps previous handle at return-from-spawn

Steps:

1. W_1 enters spawned f.
2. W_2 steals the $Main$ frame from W_1.

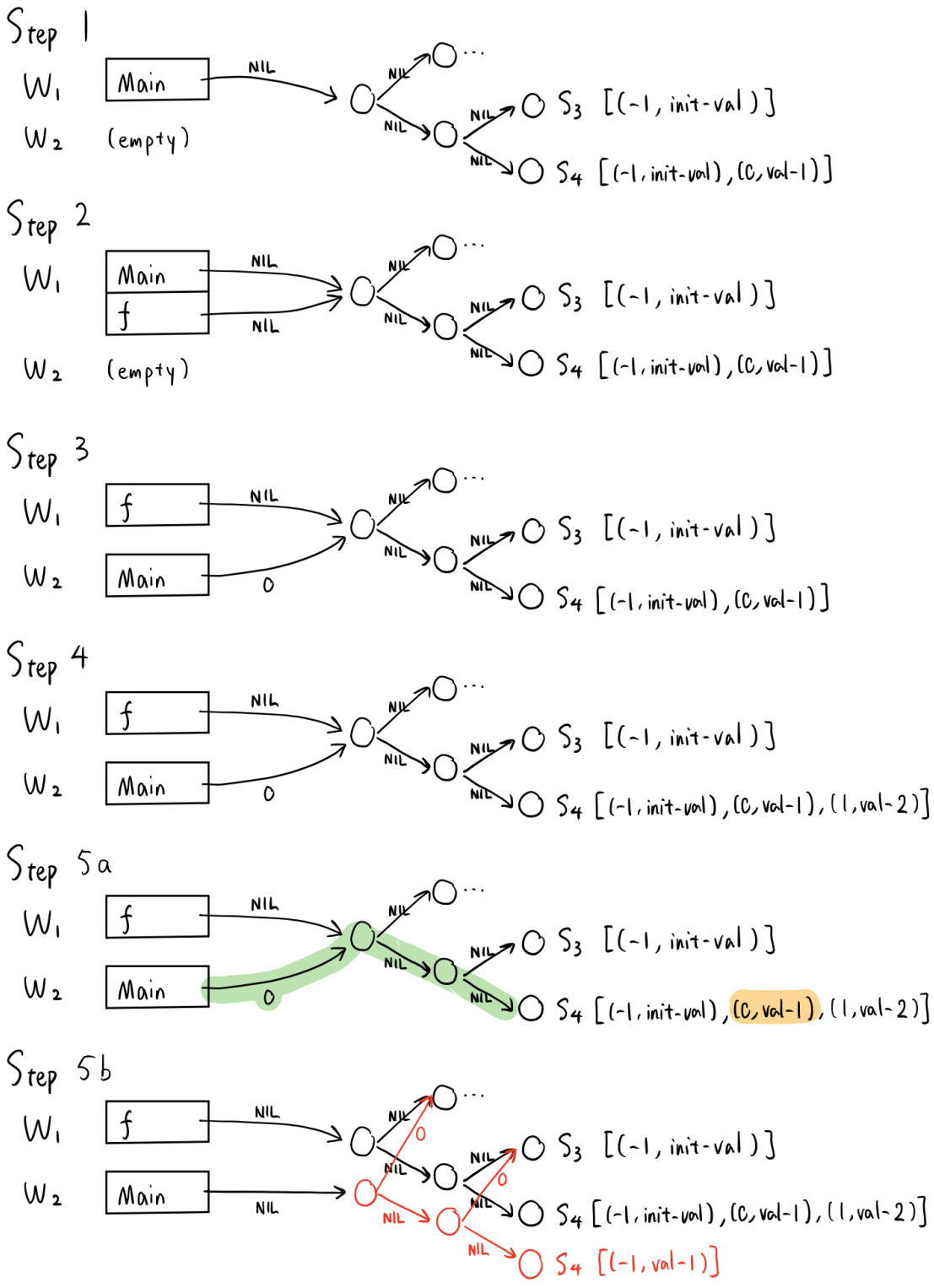

Figure 5-8: The path traversed in `depth-query` in step 5 is highlighted in green, and the entry found by the binary search is highlighted in orange.

3. W_2 enters spawned g.
4. W_2 writes `new-val` to S_4.
5. W_2 returns from g.

Figure 5-9 illustrates the state of worker W_2 after each step from 2 through 5.

The primary access to S_4 inside the spawned function g in step 4 calls `path-copy`. This operation creates a new root-to-leaf path to S_4 and updates the handle in the g frame to point to the new root. When W_2 returns from function g, it *does not* discard this root-to-leaf path by going back to using the handle stored in the $Main$ frame. Instead, in step 5, the $Main$ frame's handle is updated to be equal to the handle in the just-returned g frame.

This behavior contrasts the simple split-tree mechanism. At a return-from-spawn in the simple mechanism, the worker uses the handle stored in the parent frame. This behavior of the simple mechanism effectively discards all root-to-leaf paths created in the spawn, and undoes all writes performed in the spawn. In the stack-augmented mechanism, these writes are instead undone by explicitly popping elements from leaf stacks.

Not discarding these root-to-leaf paths is important for performance. It guarantees that expensive `path-copy` operations only happen once per splitter per steal.

Worker discards previous handle at provably good steal

Steps:

1. W_1 enters spawned f.
2. W_2 steals the $Main$ frame from W_1.
3. W_2 writes `new-val` to S_4.
4. W_2 fails to pass sync, $Main$ becomes suspended.
5. W_1 returns from f and provably-good steals $Main$.

Figure 5-10 illustrates the state of the system after steps 2 through 5 (the omitted steps are identical to those in Figure 5-7).

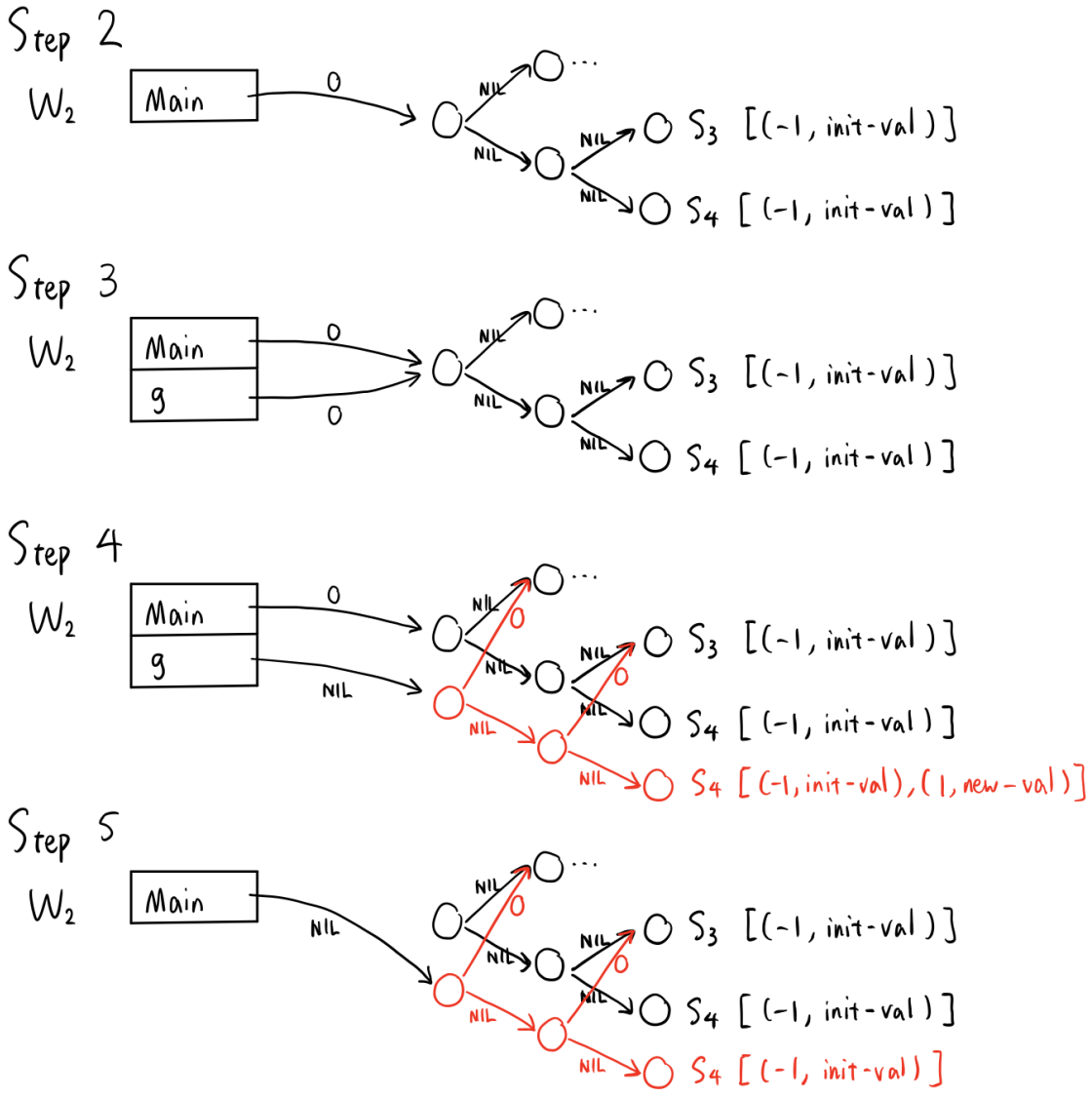

Figure 5-9: The handle in the *Main* frame is updated in step 5.

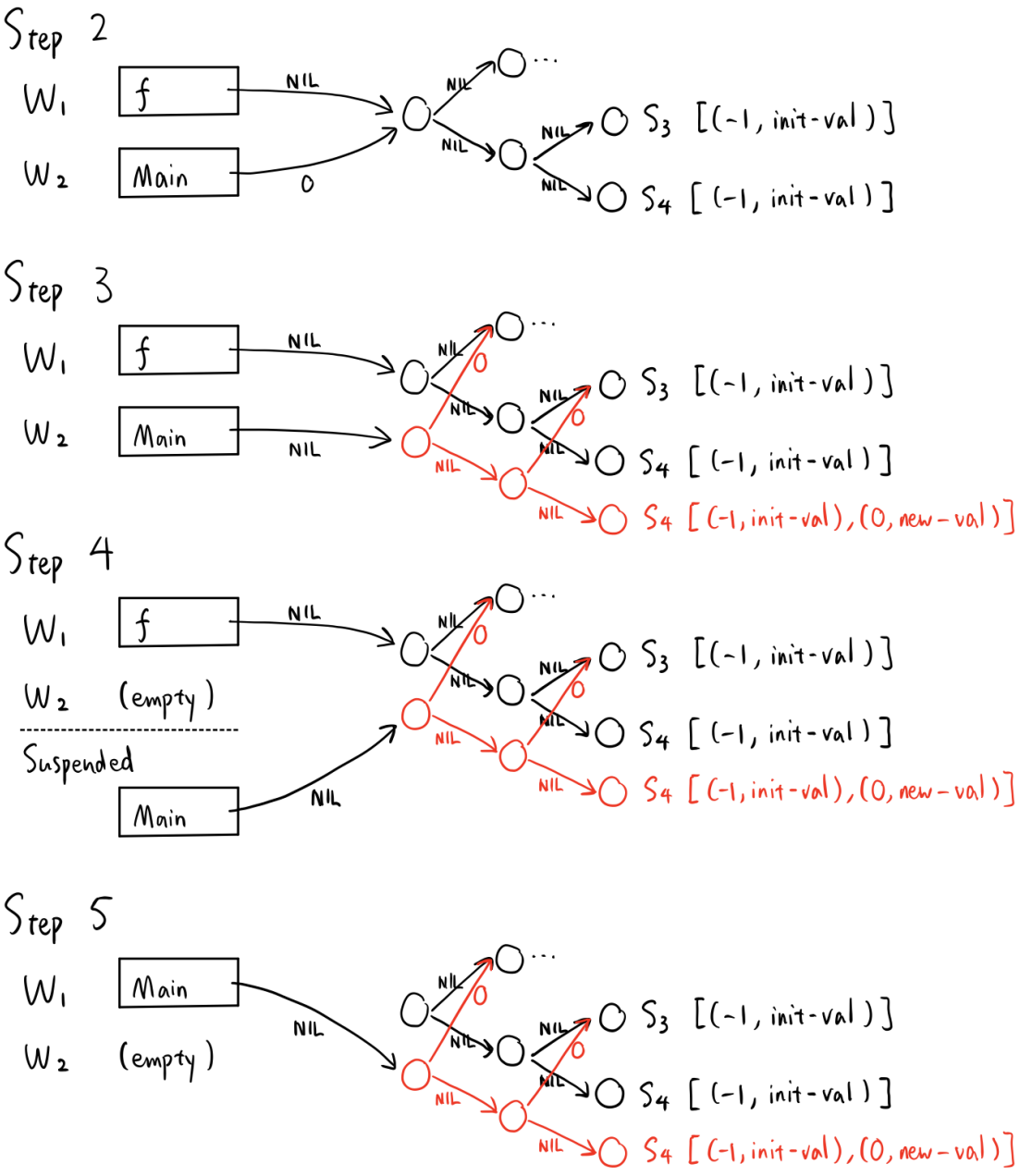

Figure 5-10: The handle in the *Main* frame is not updated in step 5.

Unlike at a normal return-from-spawn, upon a provably good steal of the parent frame, the worker *does discard* its previous handle and does not update the handle stored within the stolen frame. Changing the handle in $Main$ to point to the black root in step 5 like the handle in f is dangerous, since doing so loses any updates performed in the continuation.

Chapter 6

Explicit Memory Management in the Stack-Augmented Split-Tree Mechanism

The stack-augmented split-tree mechanism described in the previous chapter assumes automatic garbage collection. This chapter describes how the mechanism can be modified to explicitly free its allocated memory. Explicitly tracking and freeing allocated memory is necessary since Cilk, an extension of C and C++, does not support automatic garbage collection.

Section 6.1 defines key terms used in the protocol for determining responsibility for memory allocations. Section 6.2 describes the memory management protocol. Section 6.3 examines one example program in detail to illustrate subtleties in the protocol.

6.1 Definitions

This section defines key terms used to determine responsibility for deallocating memory.

Chunk: Nodes of an execution trace can be divided into chunks as follows:

- Incident nodes of a **regular**, **spawn** or **sync-continuation** edge belong to the same chunk.
- Incident nodes of a **continuation** edge belong to the same chunk if and only if this continuation is not stolen.

When all **sync-spawn** edges are removed from an execution trace, all nodes (except for the start strand) have an in-degree of 1. As a result, the execution trace becomes a tree. Chunks are the connected sub-trees that result from additionally removing all **continuation** edges corresponding to stolen continuations.

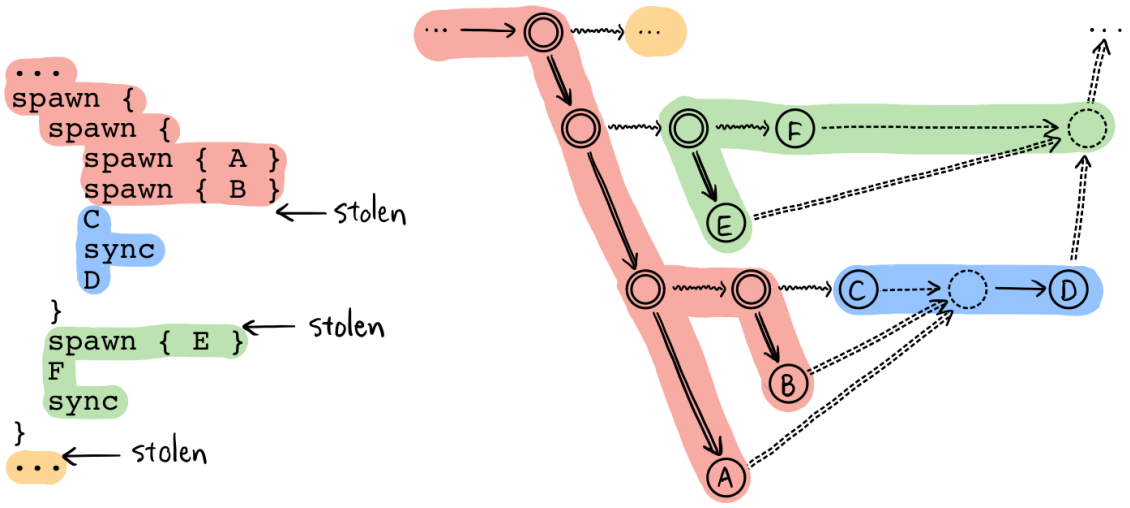

Figure 6-1: An example pseudocode snippet and execution trace, labelled with where steals occurred in the execution. Chunks are highlighted in matching colors in the pseudocode and in the trace.

Origin of a chunk: Each chunk corresponds to a particular spawn invocation called its "origin." A chunk starts within the body of its origin. In other words, the steal that started this chunk happened within the body of its origin spawn. A chunk is also said to ***originate*** from its origin.

P-footprint: A P-footprint contains the memory footprint of a single `path-copy` operation. It holds two pointers, one to the memory allocated for the root-to-leaf path created in `path-copy`, and the other to the dynamic array located in the created leaf

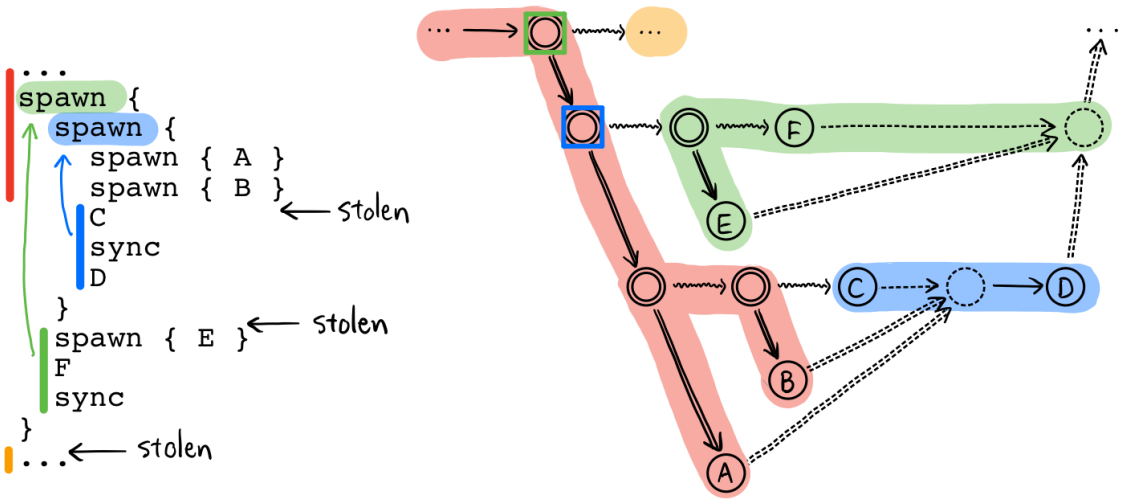

Figure 6-2: Origins of the blue and green chunks are shown with arrows and highlighting in the pseudocode, and with colored boxes in the trace. The origins of the red and orange chunks are outside of this snippet.

node. Freeing a P-footprint involves freeing both the allocated memory used to stored the root-to-leaf path, and calling destruct on the dynamic array.

P-track: The P-track of a chunk is a dynamic array of P-footprints. Every `path-copy` operation executed in this chunk appends one more P-footprint to the P-track. As a chunk is executed serially, appends to a P-track do not race.

P-log: The P-log is a dynamic array of pointers to P-tracks. The P-log located in a particular spawned frame contains pointers to the P-track of every chunk that originated from this spawn.

6.2 Protocol

The protocol ensures that memory allocations performed within a chunk are freed when the origin of this chunk returns. This is done in two stages. Allocations within a chunk are managed using a P-track, and allocations from multiple chunks with the same origin are managed using a P-log of pointers to P-tracks.

State Maintained

Every spawned function frame keeps track of a P-log. Every worker keeps track of a P-track. In addition, suspended frames temporarily hold a P-track.

Operations

Read access

If this splitter read access caused `path-copy` to be called, construct the P-footprint for this call and add it to the worker's P-track.

Write access

No-op.

Spawn

Set the P-log of the newly spawned frame to a newly created empty P-log.

Return from spawn

Consider the P-log in the just-returned spawned frame. Iterate over its contents, which are pointers to P-tracks, in parallel. For each P-track pointed to, iterate over the P-footprints contained within in parallel and free each P-footprint, then free the P-track itself. Finally, after processing all pointers to P-tracks, free the P-log.

Random steal

Create a new P-track and set it to be the worker's P-track. To the P-log located in the stolen frame, append a pointer to this new P-track.

Sync

If the sync fails and the frame becomes suspended, store the worker P-track in the suspended frame. The P-track will later be correctly reinstated.

Provably good steal of parent

Reinstate the P-track stored in the previously suspended frame as the new worker P-track. Note that the old worker P-track is not destroyed and is still accessible through a P-log.

6.3 Example

This section examines how memory is managed and freed in one example execution. The goal is to ensure understanding of the protocol.

The example is presented in two ways: as a piece of pseudocode, and as a sequence of chronological events following an execution of the pseudocode. Figures in this example include different details compared to figures in Section 5.3. Split trees are simplified and drawn without depths on edges or leaf dynamic arrays. Moreover, the example only uses two splitters as opposed to four. New details involve P-tracks and P-logs. Each P-track is represented by a color. The nodes created by a `path-copy` operation are drawn in a particular color; the P-footprint of this operation is added to the P-track represented by this color. The worker P-track is indicated to the left of the worker. Each frame contains a P-log, which is drawn as a sequence of colors.

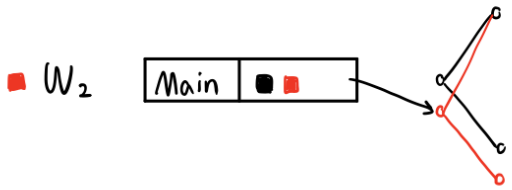

Figure 6-3: An example illustration of a worker's state. The P-log in the *Main* frame contains two P-tracks, red and black. Worker W_2's worker P-track is the red P-track. All nodes drawn in red are created in `path-copy` operations that have their P-footprints added to the red P-track.

Steps:

1. W_1 enters spawned f_1 (line 2).
2. W_1 enters spawned f_2 (line 3).

```
 1: MAIN {
 2:     spawn f1 {
 3:         spawn f2 { ... }
 4:         ...
 5:         read S2
 6:         spawn f3 {
 7:             read S1
 8:         }
 9:         ...
10:         read S1
11:         sync
12:         read S2
13:     }
14:     ...
15:     read S2
16:     read S1
17: }
```

3. W_2 steals the $Main$ frame from W_1 (line 14).
4. W_2 reads S_2 (line 15).
5. W_3 steals the f_1 frame from W_1 (line 4).
6. W_3 reads S_2 (line 5)
7. W_3 enters spawned f_3 (line 6).
8. W_4 steals the f_1 frame from W_3 (line 9).
9. W_3 reads S_1 (line 7).
10. W_3 returns from f_3 (line 8).
11. W_4 reads S_1 (line 10).
12. W_4 fails to pass sync, the f_1 frame becomes suspended (line 11).
13. W_1 returns from f_2 and provably-good steals f_1 (line 11).
14. W_1 reads S_2 (line 12).
15. W_1 returns from f_1 (line 13).
16. W_2 reads S_1 (line 16).
17. W_2 returns from $Main$, program ends.

Steps are explained in more detail, accompanied by diagrams, in the following pages.

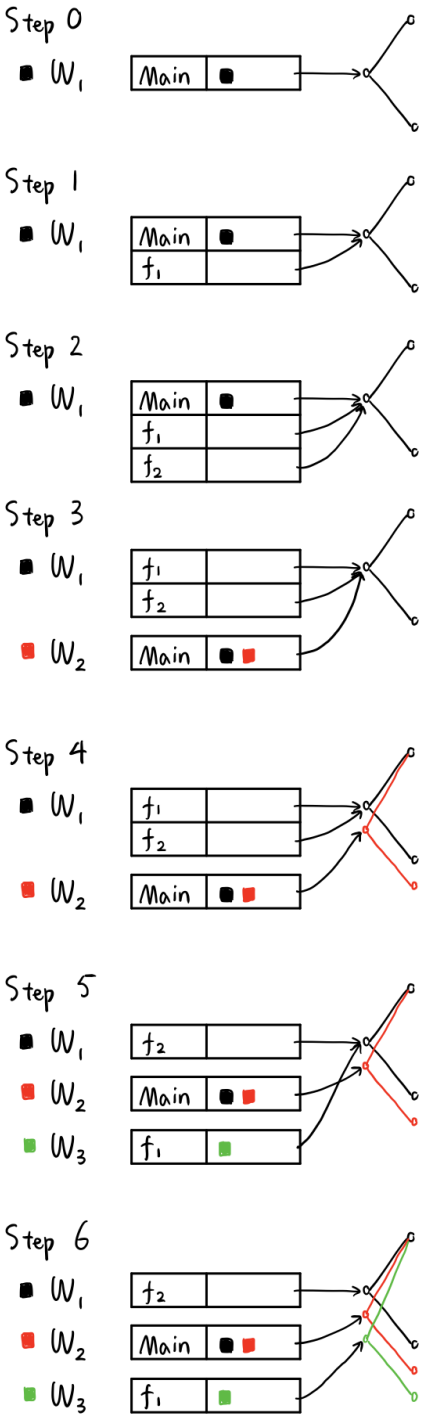

Figure 6-4

0. Program starts, W_1's deque contains the $Main$ frame.

   The P-track of C_{init} is somewhat special — no `path-copy` operations is called in C_{init}, so there are no new memory allocations for this P-track to manage. The nodes in the initial split tree (drawn in black) are instead considered to be the responsibility of the P-track of C_{init}.

1. W_1 enters spawned f_1 (line 2).

2. W_1 enters spawned f_2 (line 3).

3. W_2 steals the $Main$ frame from W_1 (line 14).

   Upon a steal, the thief starts a new chunk and is assigned a new P-track. The new P-track of W_2 is represented by red. The new P-track is added to the P-log in the stolen frame $Main$.

4. W_2 reads S_2 (line 15).

   The nodes allocated in this `path-copy` are drawn in red, as they are the responsibility of worker W_2's P-track, which is represented by red.

5. W_3 steals the f_1 frame from W_1 (line 4).

   Similar to step 3, thief W_3 is assigned a new P-track, represented by green. The new P-track is added to the P-log in the stolen frame f_1.

6. W_3 reads S_2 (line 5)

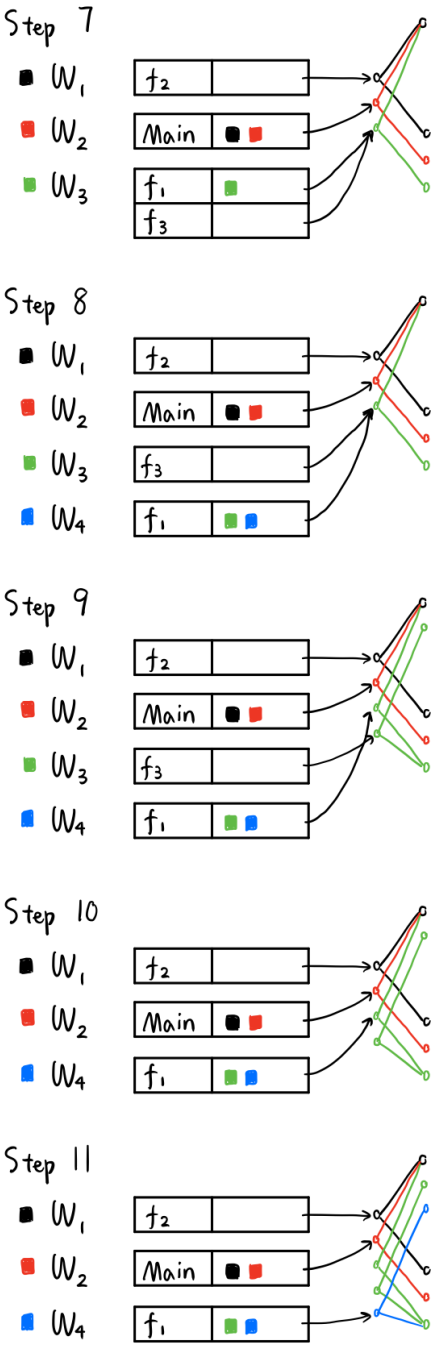

Figure 6-4

7. W_3 enters spawned f_3 (line 6).

8. W_4 steals the f_1 frame from W_3 (line 9).

   Thief W_4 is assigned a new P-track, represented by blue. The new P-track is added to the P-log in the stolen frame f_1.

9. W_3 reads S_1 (line 7).

   Nodes created by W_3 in this `path-copy` are still the responsibility of the green P-track, even though f_1, whose P-log contains the green P-track, has been stolen by W_4.

10. W_3 returns from f_3 (line 8).

    Green nodes are not freed at this step. Doing so is dangerous, as W_4 can still access some green nodes.

11. W_4 reads S_1 (line 10).

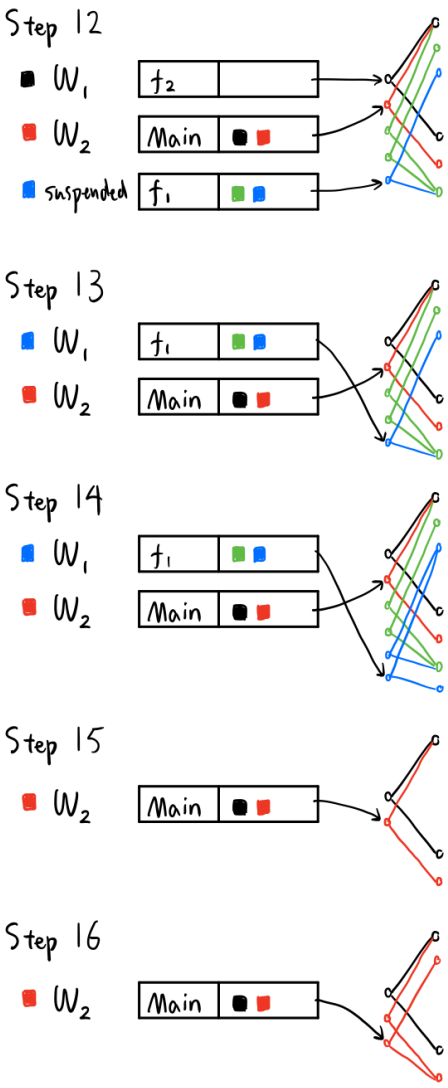

Figure 6-4

12. W_4 fails to pass sync, the f_1 frame becomes suspended (line 11).

13. W_1 returns from f_2 and provably-good steals f_1 (line 11).

    At the provably good steal, W_1 discards its previous black P-track and takes up the blue P-track stored in the previously suspended frame. The black P-track and its contents are not destroyed, merely discarded, and the black P-track is still accessible from the P-log of $Main$. Destroying black nodes at this point is dangerous, as W_2 can still access a black leaf.

14. W_1 reads S_2 (line 12).

15. W_1 returns from f_1 (line 13).

    The P-log in f_1 stores the green and blue P-tracks. When f_1 returns, all green and blue nodes are freed.

16. W_2 reads S_1 (line 16).

17. W_2 returns from $Main$, program ends.

    The P-log in $Main$ stores the black and red P-tracks. When $Main$ returns, all black and red nodes are freed.

Chapter 7

Proofs of Correctness

This chapter presents proofs that the stack-augmented split-tree mechanism is correct in three aspects — read accesses to splitter variables return correct values, the mechanism is not affected by concurrency, and memory allocations are freed thoroughly and safely.

Section 7.1 defines key terms and notations used in the proofs. Section 7.2 states and proves properties and invariants of the runtime system. Particular attention should be paid to Lemma 18 in this section, as it captures the key structural invariant of the mechanism. The next sections each focus on one aspect of correctness. Section 7.3 proves that read accesses return correct values, Section 7.4 proves that correctness holds despite possible concurrent operations, and Section 7.5 proves that all memory allocations are freed in a safe manner.

7.1 Definitions

This section defines terms and notation used in the rest of the chapter.

Definition 8. Denote by C_{init} the chunk containing the start of the execution.

Remark. C_{init} is special in several ways. It does not originate from stealing a spawn continuation, and it "owns" all of its leaf nodes right from the start, never performing `path-copy` upon accessing a splitter.

Definition 9. A chunk C is said to be the ***parent*** of chunk C' if C' was started by stealing the continuation to a spawn inside C. C' is said to be a ***child*** of C.

A chunk C is said to be an ***ancestor*** of chunk C' if there exists a sequence of chunks $C_0, \cdots, C_k$ such that $C_0 = C$, $C_k = C'$, and C_i is the parent of C_{i+1} for $0 \leq i < k$. C' is said to be a ***descendant*** of C.

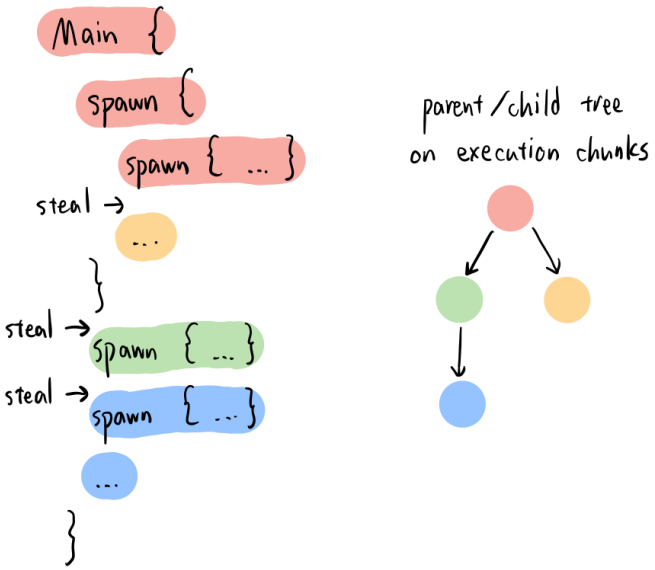

Figure 7-1: Parent/child relations among the chunks of an execution of an example code snippet. Chunks are highlighted in different colors. Each directed edge in the tree on the right points from a parent chunk to a child chunk.

Remark. The directed graph representing parent/child relations on chunks in an execution forms a rooted tree where C_{init} is the root. C is an ancestor of C' if there exists a directed path from C to C' in this graph.

Definition 10. For any chunk $C \neq C_{init}$, consider the worker that executed the very start of C. Denote by $H_{start}(C)$ its worker handle at the start of C, after the `handle-update` operation performed in the steal is complete.

Definition 11. For any chunk $C \neq C_{init}$, denote by $d_{steal}(C)$ the spawn depth of very start of C, or the spawn depth of the stolen continuation that started C.

Definition 12. For any splitter X and handle H, denote by $L_X(H)$ the leaf stack corresponding to splitter X in the split tree that H is attached to.

7.2 Basic Properties and Invariants

This section presents key properties and invariants of the runtime system's state, starting with simple lemmas and observations stated without proof, and ending with the subtle and important Lemma 18. These lemmas and observations are used in proofs in the following sections.

Lemma 13. The following statements are equivalent — that is, either all are true or none are true.

- The primary access to `splitter` in the worker's current chunk has happened.
- The value returned by `depth-query(worker handle, splitter)` equals NIL.
- The leaf stack of `splitter` reachable from the worker handle was created within the worker's current chunk.
- The worker's cache contains `splitter`.

The very start of the execution is considered to implicitly access every splitter.

Lemma 14. A leaf stack can only be modified inside the chunk where it was created.

Lemma 15. Every leaf stack's contents of `(depth, value)` are sorted by depth in increasing order, with no two entries having the same value of `depth`.

It is easy to check that, by construction, every operation in the mechanism preserves the simple invariants described in the above lemmas.

Next, we make some observations about the spawn depths of chunks.

Observation 16. Let C be any chunk. Then the spawn depths of all parts of C are greater than or equal to $d_{steal}(C)$.

Proof. When all **sync-spawn** edges are removed from the execution trace, the remaining edges form a tree. Each edge in this tree satisfies the property that the spawn

depth of its source node is less than or equal to the spawn depth of its destination node. For intuition, consider a drawing of the tree that places each node at a vertical height according to its spawn depth. A node of a larger spawn depth is drawn lower down on the page, and nodes of the same spawn depth are drawn at the same height. Then all edges of the tree in this drawing are either horizontal or point downwards. Figure 7-2 illustrates an example execution trace drawn in this manner.

Chunks are the connected sub-trees formed when all **sync-continuation** edges corresponding to stolen continuations are additionally removed from this tree. All edges in such a sub-tree are horizontal or point downwards. The first node in a chunk C is therefore the node drawn highest on the page among all nodes in C. The spawn depth of this node, $d_{steal}(C)$, is the smallest among spawn depths of all nodes in C. □

Observation 17. Let chunk C_{parent} be the parent of chunk C_{child}. Consider the spawn s in C_{parent} whose stolen continuation is the start of chunk C_{child}. Then $d_{steal}(C_{child})$ is strictly less than the spawn depth of every part of C_{parent} after s.

Proof. Thieves steal from victims starting from the top of the victim's deque. As a result, in order for the continuation to spawn s to be stolen, the continuations to those spawns in C_{parent} at higher levels must have already been stolen. Therefore, the parts of C_{parent} that take place after s are contained within the function spawned at s. All such parts have a spawn depth strictly greater than the spawn depth of s. The spawn depth of s is equal to $d_{steal}(C_{child})$. Combining these last three statements concludes the proof. □

The following lemma describes a key restriction on operations on stack-augmented split trees. This lemma captures the core of why this mechanism is correct and concurrency-safe.

Lemma 18 (Main structural lemma)**.** Let X be a splitter and let $C \neq C_{init}$ be a chunk. Let s be the spawn whose stolen continuation starts C. Then every modification of the leaf stack $L_X(H_{start}(C))$ after the execution of s happens at a spawn depth strictly greater than `depth-query`$(H_{start}(C), X)$.

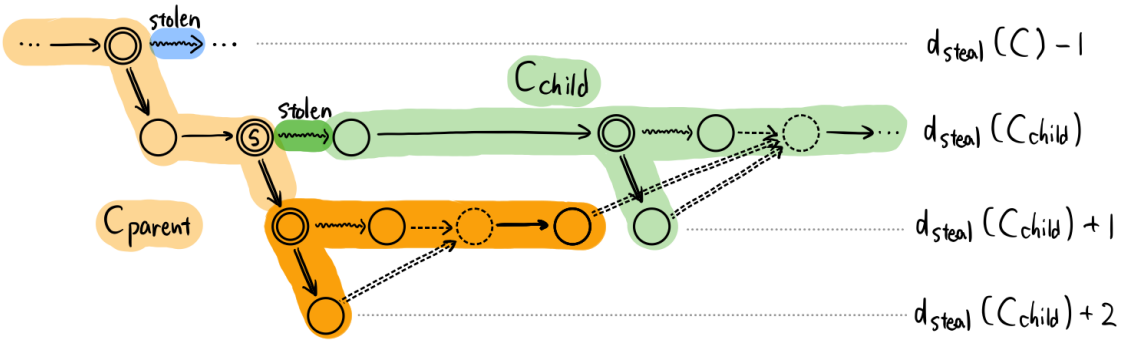

Figure 7-2: Example trace snippet illustrating Observations 16 and 17.
Observation 16: All parts of C_{child}, highlighted in green, have a spawn depth greater than or equal to $d_{steal}(C_{child})$.
Observation 17: C_{parent} is highlighted in orange, and the part of C_{parent} that happen after s is highlighted in dark orange. All nodes highlighted in dark orange have a spawn depth strictly greater than $d_{steal}(C_{child})$.

Proof. The proof proceeds by induction on chunks. We'll see that the lemma holds for children of C_{init}, and that if the lemma holds for a parent chunk, then it also holds for a child chunk.

Base case. The lemma holds when C is a child of C_{init}.

Subproof. The handle at every point in C_{init} has a depth of NIL and is attached to the initial split tree whose edges all have a depth of NIL.

The handle $H_{start}(C)$ results from calling `handle-update` on such a handle with depth NIL, and thus the depth of handle $H_{start}(C)$ is set to to d_{steal}. Since all edges of the tree that $H_{start}(C)$ is attached to have a depth of NIL, `depth-query`$(H_{start}(C), X)$ equals $d_{steal}(C)$.

The leaf stack $L_X(H_{start}(C)) = L_X(R_{start}(H_{init}))$ is created in the initialization process. By Lemma 14, it can only be modified inside C_{init}.

By Observation 17, $d_{steal}(C)$ is strictly greater than the spawn depth of all parts of C_{init} after s.

Therefore, every modification to $L_X(H_{start}(C))$ after the start of C happens at a spawn depth strictly greater than `depth-query`$(H_{start}(C), X)$. ■

Inductive step. Let chunk C_{parent} be the parent of chunk C_{child}. If the lemma holds for $C = C_{parent}$, then it holds for $C = C_{child}$.

Subproof. Let s be the spawn in C_{parent} whose stolen continuation starts C_{child}. Consider the following two cases:

- The primary access of X in C_{parent} happens after s (or never happens).

  In this case, `path-copy` on splitter X is not among the operations that evolved $H_{start}(C_{parent})$ into $H_{start}(C_{child})$. Therefore, $L_X(H_{start}(C_{parent}))$ and $L_X(H_{start}(C_{child}))$ must be the same leaf stack. By Properties 6 and 7, all other operations — calling `path-copy` on any other splitter, or calling `handle-update` — do not change the result of `depth-query`(H, X). Therefore, the value `depth-query`$(H_{start}(C_{parent}), X)$ equals `depth-query`$(H_{start}(C_{child}), X)$. The inductive hypothesis then directly implies the desired result.

- The primary access of X in C_{parent} happens before s.

  In this case, `path-copy` must have been called on splitter X before s in C_{parent}. The leaf stack $L_X(H_{start}(C_{child}))$ is thus created within C_{parent}.

  By Property 6 and the fact that `path-copy` was called on X before s in C_{parent}, `depth-query` on splitter X starting from the handle in the stolen frame equals NIL. By Property 7, the handle $H_{start}(C_{child})$ that results from calling `handle-update` on the stolen handle has `depth-query(`$H_{start}(C_{child})$`, X)` set to $d_{steal}(C_{child})$.

  By Lemma 14, the leaf stack $L_X(H_{start}(C_{child}))$ can only be modified inside C_{parent}. By Observation 17, $d_{steal}(C_{child})$ is strictly less than the spawn depth of every part of C_{parent} that executes after s. Therefore, every write after s to the leaf stack $L_X(H_{start}(C_{child}))$ happens at a spawn depth strictly greater than $d_{steal}(C_{child})$.

  Combining the conclusions of the two above paragraphs implies the desired result.

■

The graph representing parenthood on chunks forms a tree rooted at C_{init}. We have seen that the lemma holds for all direct children of C_{init}, and that if the lemma holds for a parent, then it holds for a child. Therefore, the lemma holds for every chunk in an execution other than C_{init}. □

7.3 Correctness of Accessed Values

This section proves that the stack-augmented split-tree mechanism provides the correct functionality as described in Section 2.3. The desired functionality is restated here.

Property 5. Let G be an execution trace, constructed so that every splitter read and write belongs to its own strand. Let v_{read} be a node in G performing a read to splitter X. Let v_{write} be the last node on the canonical path to v_{read} that performs a write to splitter X, if such a node exists. Then the value read in v_{read} equals the value written in v_{write}, or to the value at initialization if v_{write} does not exist.

We'll see a proof to a somewhat stronger statement:

Theorem 19. Let G be an execution trace, constructed so that every splitter write is in its own strand. Let v be any node, and let v_{write} be the last node on the canonical path to v (not including v) that performs a write to splitter X, if such a node exists. Replace the instructions in v by a read to splitter X. Then the value read in v equals the value written in v_{write}, or to the value at initialization if v_{write} does not exist.

We'll first see a proof to the following lemma, which covers a scenario that appears multiple times in the proof of Theorem 19.

Lemma 20. Let u, v be two nodes in the same chunk C, where node u executes before node v. If the primary access to splitter X in C happens at or after v (or never happens), then a read access to splitter X at nodes u and v return the same value.

Proof. Consider the sequence of `path-copy` operations that evolved the worker handle in chunk C from $H_{start}(C)$ to the worker handle in node u. Since the primary access to X does not happen by node v and thus does not happen by node u, this sequence of `path-copy` operations are only called on splitters other than X. Since no new leaf stack of X is created in this process, the leaf stack of X reachable from u is equal to $L_X(H_{start}(C))$. For the same reason, the leaf stack of X reachable from v is equal to $L_X(H_{start}(C))$.

By Property 6, the `path-copy` operations on splitters other than X cannot change the result of `depth-query` on splitter X. Thus, `depth-query` on splitter X from handle $H_{start}(C)$, from the handle in node u, and from the handle in node v all return the same non-NIL value d.

The leaf stack $L_X(H_{start}(C))$ was created within some ancestor chunk of C. By Lemma 18, the entries of $L_X(H_{start}(C))$ that correspond to a depth less than or equal to d cannot change after C starts execution.

Reading splitter X at nodes u and v involve searching the same leaf stack for the entry whose depth is the largest possible less than or equal to d. Since entries of depth less than or equal to d in this leaf stack cannot change, the leaf stacks searched at u and at v share a prefix up to the target entry. Therefore, the same entry is found in the two searches, and the same value returned at the two read accesses. □

Proof of Theorem 19. Proceed by induction on nodes of the execution trace. We'll see that the theorem holds for the start strand, and that if the theorem holds for the source node of a regular, spawn, sync-continuation, or continuation edge in the trace, then it also holds for the destination node.

Base case. The theorem holds for when v is the start strand.

Subproof. Trivially true by construction. ■

Inductive case 1. Let uv be a regular or sync-continuation edge where node u performs a write to splitter X. If the theorem holds for node u, then it holds for v.

Subproof. Nodes u and v belong to the same chunk, and node u is executed immediately before v in this chunk.

After the execution of u, splitter X is guaranteed to be cached, with the last entry of the cached leaf stack reflecting the newly written value. By Lemma 14, this leaf stack cannot be modified by some other worker between the execution of u and v. The read access at node v examines the last entry in the cached leaf stack and correctly returns the value written in u.

Note that this case does not require the inductive hypothesis. ■

In all cases below, the last write access to splitter X is the same on the canonical paths to u and to v. We'll show that the values accessed at u and v are the same.

Inductive case 2. Let uv be a regular, spawn, or sync-continuation edge where node u does not perform a write to splitter X. If the theorem holds for node u, then it holds for node v.

Subproof. Nodes u and v belong to the same chunk C, and node u is executed immediately before v in this chunk. Considering the following cases:

- The primary access of X in C happens before u.

  The same leaf stack created within C is reachable from u and from v. By Lemma 14, this leaf stack is not modified between u and v. The read accesses at u and v thus examine the last entry of the identical leaf stacks, returning the same value.

- The primary access of X in C happens at u.

  The read access at u creates a new leaf stack with only a single entry that contains the value read at u. The read access at v examines the last, and only, entry in this leaf stack, returning the same value as the one read at u.

- The primary access of X in C happens at or after v (or never happens).

  Lemma 20 finishes this case.

■

Inductive case 3. Let uv be a continuation edge where the continuation was not stolen. If the theorem holds for node u, then it holds for node v.

Subproof. Since thieves steal starting at the top of the victim's deque, the fact that continuation uv is not stolen implies that no continuation is stolen within the function spawned at u. Therefore, the same worker executes u, the spawned function, then v, within the same chunk C.

Consider the following cases:

- The primary access of X in C happens before u.

  The same leaf stack created within C is reachable from u and from v. Every splitter write that happens in the spawned function is undone (through popping entries from the stack) by the time the spawned function returns. The read accesses at u and v thus examine the last entry of the identical leaf stacks, returning the same value.

- The primary access of X in C happens after u, and before v.

  Let the primary access of X in C take place at node w inside the function spawned at u. By Lemma 20, the same value is read at u and at w.

  The execution of node w performs a `path-copy` operation and creates a new leaf stack with a single entry that contains the value read in w. All other entries appended to this leaf stack in the spawned function are popped by the time that the spawned function returns. The read access at v examines the last entry, which is also the only entry, in this leaf stack, returning the value read in w. Since the values read in w is the same as that read in u, the read accesses at u and v return the same value.

- The primary access of X in C happens at or after v.

  Lemma 20 finishes this case.

■

Inductive case 4. Let uv be a continuation edge where the continuation was stolen. If the theorem holds for node u, then it holds for node v.

Subproof. Let the chunk containing node u be C, and the chunk containing node v be C'. Note that the same leaf stack of X is reachable from u and v.

Consider the following cases:

- The primary access to splitter X in C happens before u.

  Since the primary access to X, which calls `path-copy` on X, has taken place by node u, we know that `depth-query` on X at node u returns NIL by Property 6. By Lemma 18, those entries in the leaf stack of depth less than or equal to d cannot be changed after u executes. The contents of the leaf stack at u is therefore a prefix of the contents of the leaf stack at node v. The entry found in the read access at v, the entry whose depth is the largest possible less than or equal to d, is the last entry in this prefix. The read access at u takes the last entry in its leaf stack, which is the same entry. The read accesses at u and v thus return the same value.

- The primary access to splitter X in C happens after u (or never happens).

  The proof for this case is similar to the proof to Lemma 20. Since the primary access to C has not taken place by u, `depth-query` on X at u returns some non-NIL value d. By Property 7, `depth-query` on X at node u returns the same value d. By Lemma 18, those entries in the leaf stack of depth less than or equal to d cannot be changed after u executes.

  Reading splitter X at nodes u and v involve searching the same leaf stack for the entry whose depth is the largest possible less than or equal to d, and the contents searched at u and v share a prefix up to the target entry. Therefore, the same entry is found in the two searches, and the same value returned at the two read accesses.

■

When all sync-spawn edges are removed from the execution trace, the resulting graph is a tree. We've seen that the theorem holds for the root of the tree, and that

if the theorem holds for a parent node then it holds for its child nodes. Therefore, the theorem holds for all nodes in the trace.

□

7.4 Concurrency Safety

This section shows that the mechanism behaves correctly despite potential concurrent operations from multiple workers.

The problem of interest is concurrent accesses or modifications to the same leaf stack. A dynamic array uses a reader-write lock, and thus the mechanism avoids the most direct type of races on a leaf stack. We need to additionally ensure that a reader or writer making a sequence of operations to a leaf stack is not impacted by concurrent writes from other workers interleaved into the sequence.

Firstly, observe that concurrent writes to the same leaf stack cannot happen. Lemma 14 implies that only one chunk is able to modify any leaf stack, and a chunk is executed serially.

The remaining question is whether it is safe for reads to happen concurrently with modifications from a writer. Such reads happen as a part of the binary search performed at a primary access. By Lemma 18, the target of the binary search, as well as all entries before it in the leaf stack, cannot be modified by the writer. It is possible for the part of the stack after the target entry to change between consecutive reads in the same binary search — for instance, the writer may perform many `pop`s in a row, which shortens the stack to the point where the reader's next read is at an index past the end of the stack. This turns out to not be a problem. An "invalid value" from reading at an index past the end of the stack implies that this index is larger than the index of the target entry. Thus, as long as the reader treats "invalid values" as if a too-large value was read, the binary search will complete successfully and return the correct target value.

7.5 Memory Safety

This section shows that the mechanism manages memory safety — all memory is freed by the end of the program, and no memory is freed when still potentially useful.

We're interested only in the memory allocated in calls to `path-copy`. Other memory allocations, such as caches and records, are freed in a very straightforward fashion.

We start by looking at a simple lemma that is evident by the construction of the memory management protocol.

Lemma 21. Every P-footprint belongs to exactly one P-track, and every P-track is pointed to in exactly one P-log.

At a return-from-spawn, the mechanism takes the P-log in the returning frame and frees all P-footprints in P-tracks pointed to by this P-log. Therefore, every P-footprint is freed exactly once by the end of the program, once every spawned frame has returned.

It remains to be shown that a P-footprint is not freed until it can no longer be accessed.

Lemma 22. Let C be a chunk with origin s. A P-footprint created within C is inaccessible after s returns.

Proof. Note that by the time s returns, all descendant chunks of C, along with C itself, have completed. A P-footprint created in C can only be accessed from C and its descendant chunks. Therefore, this P-footprint is inaccessible once s returns. □

It is *not true* that a P-footprint created within some spawn is inaccessible after this spawn returns, or that a P-footprint created within some chunk C is inaccessible after C ends. If the continuation to a spawn was not stolen, then a P-footprint created within this spawn may continue to be used after this spawn, as illustrated in example **worker keeps previous handle at return-from-spawn** in Section 5.3. A P-footprint created within a chunk may still be used after this chunk ends if a child of this chunk is still ongoing, as illustrated in step 10 of the example in Section 6.3.

The mechanism specifies that a P-footprint is freed exactly at the return of the origin of the chunk where the P-footprint is created. Therefore, memory is managed safely and not freed until no longer used.

Chapter 8

Analysis of Theoretical Overhead

This chapter analyzes the theoretical performance impact of using the stack-augmented split-tree mechanism to support splitters. We'll see how the *real* runtime of an execution can be expressed based on performance measures of the execution trace from the *user's perspective*. We'll find that in the theoretical worst case, using n splitters causes the real running time of a program to behave as if the parallelism of the user program is impacted by a factor of roughly $\tilde{O}(n)$.

Defining the User Trace and the Runtime Trace

We start by looking at two key concepts, the user trace and the runtime trace. The user trace represents an execution trace from the user's perspective, and the runtime trace represents what actually took place in the execution.

The ***user trace*** of a computation, G_{user}, is the execution trace that does not account for any overheads from splitter operations. In particular, every splitter access in the user trace costs constant time for some sufficiently large constant, and a return-from-spawn takes constant time. From the perspective of the user, who only knows about instructions explicitly invoked in the program, the user trace is the correct trace.

The ***runtime trace*** of a computation, $G_{runtime}$, is the execution trace that reflects what actually took place. In the runtime trace, some splitter accesses take more than

constant time due to overheads from `path-copy`, and some return-from-spawns take more than constant time due to cleanup costs associated with record-keeping and P-log management.

For ease of analysis, G_{user} is constructed such that every splitter access or write is in its own strand, and every return-from-spawn is in its own strand at the end of the returning function. Note that G_{user} and $G_{runtime}$ have nearly identical structures. $G_{runtime}$ can be constructed from G_{user} by splicing in substructures to replace some return-from-spawn nodes, changing the cost of some splitter operation strand nodes, and retaining the rest of the graph without change.

We'll express the true running time, or $T_P(G_{runtime})$, in terms of performance measures visible to the user — the work and span of G_{user}. By Lemma 2, $T_P(G_{runtime}) = T_1(G_{runtime})/P + (T_\infty(G_{runtime}))$ in expectation. Therefore, we need to bound the work and span of $G_{runtime}$ in terms of those of G_{user}.

In the rest of the analysis, n denotes the number of splitters used in the computation.

Work and Span of the Runtime Trace

We start by accounting for the costs of splitter accesses. As defined before, D denotes the maximum depth of nested spawns.

Lemma 23. The primary access of a splitter in a chunk costs $O(\log n + \log D)$. All secondary accesses cost $\Theta(1)$.

Proof. By Lemma 13, a splitter exists in a worker's cache if and only if the primary access has already taken place. A secondary access extracts the cached leaf stack and returns the last value in the stack, which takes $\Theta(1)$ time. A primary access takes $O(\log n)$ time to perform `depth-query` and `path-copy`, as well as $O(D)$ time to search through a leaf stack containing up to one entry for each possible distinct spawn depth. □

Each splitter access in G_{user} therefore corresponds to a node in $G_{runtime}$ whose

cost is either $O(\log n + \log D)$ or $\Theta(1)$.

Next, we consider the effect of record-keeping at return-from-spawns. Upon returning from a spawn, the mechanism undoes all writes performed in the spawn by popping entries from leaf stacks. If m splitters have been written to inside the spawn, then this cleanup takes $O(m)$ work and $O(\log m)$ span, as the cleanup happens via a parallel loop over the record.

The number of splitters written to inside a returning spawn can be crudely bounded by the total number of splitters. Therefore, each return-from-spawn node in G_{user}, after taking into account record-keeping costs, corresponds to a substructure in $G_{runtime}$ with up to $O(\log n)$ span.

Finally, we consider the cost of managing logs. Upon returning from a spawn, the size of the P-log in the returning frame is equal to the number of steals that occurred inside the body of the spawn. If k steals happened inside the body of the spawn, iterating over the P-log in parallel takes $O(k)$ work and $O(\log k)$ span. For each P-track iterated in the P-log, cleaning up the P-track involves parallel iterating over its contents, and the cost of this operation depends on the length of the P-track.

Each return-from-spawn node in G_{user}, after taking into account both record-keeping and log-management, corresponds to a substructure in $G_{runtime}$ with up to $O(\log n + \log k)$ span, where k is the number of steals that occurred in the body of the returning spawn. The $\log n$ term results from the fact that the length of a P-track can be crudely bounded by n.

Given how each of these overheads locally transform G_{user} into $G_{runtime}$, we consider the global impact of these overheads on running time by bounding the work and span of $G_{runtime}$.

Lemma 24. In expectation,

$$T_\infty(G_{runtime}) = O(\log n + \log D + \log P)T_\infty(G_{user}).$$

Proof. Consider the most expensive path through $G_{runtime}$ and the corresponding path through G_{user}. Let v be a node on the path through G_{user}. We analyze the

length of the path through $G_{runtime}$ by considering how v is transformed into the corresponding node or path in $G_{runtime}$.

Case 1: v is not a splitter access or a return-from-spawn.

v is unchanged between G_{user} and $G_{runtime}$.

Case 2: v is a splitter access.

v corresponds to a node of cost up to $O(\log n + \log D)$ in $G_{runtime}$.

Case 3: v is a return-from-spawn.

v corresponds to a path of cost up to $O(\log n + \log k)$ in $G_{runtime}$, where k is the number of steals that occurred in the body of the returning spawn.

Temporarily ignoring the $O(\log k)$ cost of return-from-spawn nodes, the cost of the path through $G_{runtime}$ is at most $O(\log n + \log D)$ times the cost of the path through G_{user}, as each individual node on the path from G_{user} can increase in cost by a factor of at most $O(\log n + \log D)$. Since the cost of a path through G_{user} is upper bounded by $T_\infty(G_{user})$, we find that the cost of the path through $G_{runtime}$ is bounded by $O(\log n + \log D)T_\infty(G_{user})$.

Analyzing the impact of the $O(\log k)$ overhead at return-from-spawn nodes takes more finesse, as there is no useful bound on this overhead in any one return-from-spawn node. Instead, we'll see how to bound the *total* of these $O(\log k)$ overheads on the entire path through $G_{runtime}$.

Let the number of return-from-spawn nodes along this path in $G_{runtime}$ be l. Clearly, $l \leq T_\infty(G_{user})$. Let the number of steals that happened in the body of each of these spawns be $k_1, k_2, ..., k_l$. To be precise in the following algebra and avoid undefined behavior if $k_i = 0$, the span of iterating over a P-log of length k is considered to be $\log(1 + k)$. The total overhead on this path from iterating over P-logs is thus $\sum_{i=1}^{l} \log(1 + k_i)$.

Denote the total number of steals in the execution of $G_{runtime}$ by S. Each steal can contribute to at most one k_i, and thus $\sum_{i=1}^{l} k_i \leq S$. By Lemma 1, the expected number of steals $\mathbb{E}[S] = O(P \cdot T_\infty(G_{runtime}))$.

We now upper bound the overhead $\sum_{i=1}^{l} \log(1+k_i)$. Firstly, we can assume that $l = T_\infty(G_{user})$; if l is in fact smaller than $T_\infty(G_{user})$, we may set the value of all extra variables $k_{l+1}, ..., k_{T_\infty(G_{user})}$ to 0 without impacting either the restriction on $\sum k_i$ or the quantity being bounded. Applying Jensen's inequality, we find that under the restriction $\sum_{i=1}^{T_\infty(G_{user})} k_i \leq S$, the expression $\sum_{i=1}^{T_\infty(G_{user})} \log(1+k_i)$ is maximized to a value of $T_\infty(G_{user}) \log(1 + S/T_\infty(G_{user}))$, achieved when all k_i are equal. Applying Jensen's inequality again on the expectation of the log of a random variable, we derive that

$$\begin{aligned}
\mathbb{E}[\sum_{i=1}^{l} \log(1+k_i)] &\leq \mathbb{E}[T_\infty(G_{user}) \log(1 + S/T_\infty(G_{user}))] \\
&\leq T_\infty(G_{user}) \log(\mathbb{E}[1 + S/T_\infty(G_{user})]) \\
&\leq T_\infty(G_{user}) \log\left(1 + O\left(\frac{P \cdot T_\infty(G_{runtime})}{T_\infty(G_{user})}\right)\right) \\
&= T_\infty(G_{user}) \left(\log P + \log\left(\frac{T_\infty(G_{runtime})}{T_\infty(G_{user})}\right) + O(1)\right).
\end{aligned}$$

Combining the bound on these $\log k$ overheads and our previous derivation about all other overheads, we find that the cost of the most expensive path through $G_{runtime}$, equivalently the span of $G_{runtime}$, may be upped bounded in expectation as

$$\mathbb{E}[T_\infty(G_{runtime})] \leq \left(O(\log n + \log D) + \log P + \log\left(\frac{T_\infty(G_{runtime})}{T_\infty(G_{user})}\right)\right) T_\infty(G_{user}).$$

Dividing both sides by $T_\infty(G_{user})$ and rearranging,

$$\mathbb{E}\left[\frac{T_\infty(G_{runtime})}{T_\infty(G_{user})} - \log\left(\frac{T_\infty(G_{runtime})}{T_\infty(G_{user})}\right)\right] \leq O(\log n + \log D) + \log P.$$

With all overheads taken into account, the expected value of $T_\infty(G_{runtime})/T_\infty(G_{user})$ is on the order of $O(\log n + \log D + \log P)$, which implies the lemma. □

Lemma 25. In expectation,

$$T_1(G_{runtime}) = T_1(G_{user}) + P \cdot O(n(\log n + \log D) T_\infty(G_{runtime})).$$

Proof. First, note that overheads for record keeping and for iterating over the contents of P-tracks can be amortized against splitter accesses. Therefore, assuming that the costs of splitter access nodes are set to a sufficiently large constant in G_{user}, we do not need to account for these overheads when considering the work of $G_{runtime}$.

We focus on the overhead from non-constant cost splitter operations, and from append to and iterating over P-logs. Each steal in the execution causes a constant amount of extra work from creating a P-track, adding it to a P-log, and later iterating over the contents of a P-log. More importantly, each steal can cause up to n expensive splitter primary accesses, each with a cost of $O(\log n + \log D)$. Therefore, each steal can cause an increase in work of $O(n(\log n + \log D))$.

In expectation, the number of steals that occur in the execution of $G_{runtime}$ is $O(P \cdot T_\infty(G_{runtime}))$ by Lemma 1. Combining these facts yields the lemma. □

Running time of a Program Using Splitters

To conclude this analysis, we look at how the true running time of an execution that uses splitters can be bounded in terms of performance measures known to the programmer.

Theorem 26. In expectation,

$$T_P(G_{runtime}) = \frac{T_1(G_{user})}{P} + O(\tau \cdot T_\infty(G_{user}))$$

where $\tau = n(\log n + \log D)(\log n + \log D + \log P)$.

Proof. By Lemma 2, $T_P(G_{runtime}) = T_1(G_{runtime})/P + O(T_\infty(G_{runtime}))$ in expectation. Combining this result with Lemmas 24 and 25 yields the theorem. □

Recall that without overheads from the splitter mechanism,

$$T_P(G_{user}) = \frac{T_1(G_{user})}{P} + O(T_\infty(G_{user})).$$

Comparing the two bounds shows us that splitters may theoretically cause the effective parallelism of a program to drop by a factor of τ. Nonetheless, this theoretical worst-case overhead analyzed here may not be realized in practice.

Chapter 9

Evaluation of the Commutative Mechanism for Cilk Reducer Hyperobjects

9.1 Introduction

Many algorithms and programs involve accumulation of values onto some common variable. For instance, taking the sum of an array of integers involves repeatedly updating a variable representing the partial result. Figure 9-1 illustrates a more complex example that involves collecting nodes into a linked list during a binary tree traversal.

These common variables inhibit parallelization of a program, since they cause "race conditions" in parts of the program that could otherwise run in parallel. A ***determinacy race*** [38] occurs when a thread updates a memory location while another thread concurrently accesses the same location, producing nondeterministic behavior. For instance, in the program in Figure 9-1, the left and right branches of a tree node could be walked in parallel if not for the common variable `l`. This parallelization, illustrated in Figure 9-2, contains a determinacy race on variable `l` due to potential concurrent `push` operations executed on different nodes `x`. Depending

```
struct llist_t l;
void walk(node_t *x) {
  if (x) {
    if (has_property(x)) {
      push(l, x);
    }
    walk(x->left);
    walk(x->right);
  }
}
```

Figure 9-1: Serial C code collecting nodes of a binary tree that satisfy a particular property into the linked list `l`. Figure adapted from [41].

on the exact implementation of `push`, these concurrent operations could cause updates to be lost or leave `l` in an inconsistent state.

```
struct llist_t l;
void walk(node_t *x) {
  if (x) {
    if (has_property(x)) {
      push(l, x);
    }
    cilk_spawn walk(x->left);
    walk(x->right);
    cilk_sync;
  }
}
```

Figure 9-2: An incorrect Cilk parallelization of Figure 9-1, with a determinacy race on the variable `l`.

Cilk ***reducer hyperobjects*** provides a way to avoid determinacy races in such programs. A reducer hyperobject helps concurrent threads coordinate updates to a shared variable by providing different threads with different "views" of the variable, and a thread may access and modify its view without synchronizing with other threads. Reducer hyperobjects can avoid the performance, scalability, and correctness problems caused by the traditional solution of using locks around critical regions.

Formally, a reducer can be described by an algebraic monoid $(T, \otimes, e)$, where T is a set and $\otimes$ is a binary operation over T with identity e. In the context of Cilk,

a reducer is defined by an underlying data type, a reduction function for merging two views, and an identity function for producing a new view. For instance, Figure 9-1 can be correctly and efficiently parallelized using a linked list reducer with list concatenation as its reduction operation.

```
struct llist_t __reducer__((list_concat, list_new)) l;
void walk(node_t *x) {
  if (x) {
    if (has_property(x)) {
      push(l, x);
    }
    cilk_spawn walk(x->left);
    walk(x->right);
    cilk_sync;
  }
}
```

Figure 9-3: A correct Cilk parallelization of Figure 9-1 using a linked list reducer.

The existing implementation of reducer hyperobjects in Cilk follows the mechanism first described by Frigo et al. in [41] and later further analyzed and discussed in [53–55]. This mechanism satisfies the semantic guarantee that if the reduction operation is associative, then the final value of the reducer is deterministic and identical to if all updates had occurred serially. Commutativity is not required. For instance, the linked list concatenation operation is associative, but not commutative as concatenating two lists in reverse order results in differently-ordered elements. In Figure 9-3, the final contents of the linked list reducer l is guaranteed to be identical to what results from a serial walk of the tree.

This semantic property is stronger than what is offered by reductions in some other concurrency platforms. As an example, the reduction mechanism in OpenMP does not specify the order in which values are combined, and a reduction operation must be both commutative and associative in order to ensure that the result is the same as what the serial code produces [62, p. 299].

Attaining this strong semantic guarantee comes with a cost in performance, however. A non-constant cost reducer can have a surprisingly large effect on the paral-

lelism, and thus the running time, of a program. As analyzed in [55], a reducer with a τ-cost reduction can effectively decrease the parallelism of a program by a factor of τ^2.

In many cases, it is not necessary to maintain the order of reductions. Many common reductions are naturally commutative, including integer sum, max, and min, as well as bitwise AND, OR, and XOR. Even linked list concatenation is logically commutative if one only cares for the contents of the list but not the order of its elements. When the reduction operation is commutative, we can devise alternative mechanisms with potentially better performance guarantees.

This chapter investigates a new mechanism for supporting reducer hyperobjects in the Cilk runtime, called the ***commutative mechanism***. As its name implies, the commutative mechanism requires that the reduction operation be commutative in addition to associative. Section 9.2 reviews the existing mechanism, called the ***associative mechanism*** for ease of reference. Section 9.3 describes the new commutative mechanism. Section 9.4 highlights the differences in their semantics and the theoretical performance guarantees, noting that the commutative mechanism lacks some desirable semantic properties but has a lower theoretical impact on the parallelism of a program. Section 9.5 evaluates the two mechanisms in practice by comparing their performance on several microbenchmarks. We'll see that the commutative mechanism performs better than the associative mechanism by 1%-30% among the realistic benchmarks considered.

9.2 The Existing Associative Mechanism

This section reviews how the associative mechanism works as described in by Frigo et al. in [41].

The associative mechanism creates, destroys, and combines views of reducers dynamically throughout the execution of a program. Ownership of views is defined in terms of ***strands***. As defined in Section 2.1, a strand is a sequence of instructions containing no parallel flow control.

At every point during the execution of a program, each view of the reducer is ***owned*** by one strand. At parallel control points, views evolve as follows:

- When a reducer is first created, the strand that creates the reducer owns an initial view of the reducer with the identity value e.

- At a **spawn**, two new strands are created: the child strand C corresponding to the spawned function, and the parent strand P corresponding to the continuation. The child strand inherits the view owned by the strand before the **spawn**, and the parent strand owns a new view with the identity value e.

- At a return from **spawn**, the view owned by the child x_C is reduced with the view owned by the parent x_P. To be precise, x_C is updated to a value of $x_C \otimes x_P$, where $\otimes$ is the reduction operation. x_P is then destroyed, and the parent strand becomes the new owner of x_C.

There are two key optimizations related to the identity view. Firstly, views are created lazily, and a new identity view only needs to be created when first accessed. Secondly, reducing the identity view with any other view is a trivial operation and does not require actually paying the cost of the reduction operation. Due to these two optimizations, as long as the continuation to a spawned function is not stolen, this spawn and return-from-spawn do not result in the initialization and reduction of a new view. Instead, the same view simply changes ownership from the parent strand to the child strand at a **spawn**, and from the child strand back to the parent strand at the return from this **spawn**. During a serial execution, for instance, the entire program is executed using only a single view, incurring no overheads for initializations or reductions. A new view is only created upon the first access after a steal, and the initialization and reduction overheads only occur due to successful steals.

By construction, reductions happen in the serial left-to-right order. Moreover, reductions happen eagerly. Whenever a strand completes, the view owned by the strand is reduced with the adjacent left and right views if possible. By the point that a program passes a **sync**, all strands before the sync must have completed and all

necessary reductions taken place. The programmer does not need to explicitly invoke the reduction operation.

9.3 The Commutative Mechanism

In the commutative mechanism, views are largely static. Ownership of views is defined based on workers. An identity view is created upon a worker's first access to a reducer, and all future operations performed by this worker to this reducer modify this same view. Throughout the execution of a program, a worker performs a number of steals and executes code in multiple non-adjacent sections of the program; as a result, modifications performed on each view are out-of-order compared to a serial execution.

Unlike in the associative mechanism, reductions do not happen eagerly and implicitly. Reductions are triggered by an explicit invocation of `reducer_merge`, which accumulates all workers' views onto the view of the worker W_i that performs the merge. `reducer_merge` iterates over the views of all other workers W_j in some arbitrary order, updates the view of W_i to the result of reducing the views of W_i and W_j, and resets the view of W_j to the identity value e.

Updates and reductions do not happen in the serial left-to-right order for two reasons: Modifications performed on a single view are out-of-order, and reductions of views take place in arbitrary order. Therefore, the reduction operation must be both commutative and associative in order to ensure that the final result is deterministic and equal to what the serial projection of the program produces.

9.4 Comparison of Semantics and Theoretical Performance Impact

Previous sections have already described the most major difference between the two mechanisms, which is that commutativity of the reduction operation is required to obtain a deterministic result under the commutative mechanism. This section highlights

some other important differences between the two mechanisms.

Under the associative mechanism, reducer views obey the ***peer-set semantics*** as defined in [53]. The peer set of a strand u in a trace consists of those strands that are neither ancestors nor descendants of u. Peer-set semantics guarantees that, given two strands with the same peer set, the difference between the views of a reducer at these two strands is fixed. The peer-set semantics can be stated formally as follows.

Definition 27. Let G be an execution trace, and let u and v be two strands in G with the same peer set. Consider a serial walk of G, and let $a_1, \cdots, a_k$ denote the updates to a reducer x after the start of strand u and before the start of strand v. Let $x(u)$ and $x(v)$ denote the views of x at the start of strands u and v, respectively. Then $x(v) = x(u) \otimes a_1 \otimes a_2 \otimes \cdots \otimes a_k$.

As an example, consider the program shown in Figure 9-4, which recursively counts the number of nodes in a binary tree using an integer sum reducer. The lines retrieving the values `count_before` and `count_after` within the same instance of `walk` have the same peer set. By peer-set semantics, if the `count` reducer is implemented using the associative mechanism, then the views of `count` at these two points in the program differ by exactly the updates that occur in the lines in-between. Therefore, the difference between `count_after` and `count_before` is guaranteed to equal the number of nodes within the subtree rooted at node `x`.

```
int __reducer__((opadd, opzero)) count;
void walk(node_t *x) {
  if (x) {
    int count_before = count;
    count++;
    cilk_spawn walk(x->left);
    walk(x->right);
    cilk_sync;
    int count_after = count;
  }
}
```

Figure 9-4: A parallel program using an integer sum reducer for counting nodes in a binary tree.

The commutative mechanism does not satisfy peer-set semantics. As the integer addition operation is commutative, using the commutative mechanism in Figure 9-4 gives a correct final value for `count`. However, there is no guarantee on how the intermediate values of `count_before` and `count_after` are related.

In addition to semantic differences, the two mechanisms differ in what theoretical bounds can be derived on their performance impact.

Under the associative mechanism, non-constant cost reductions can have a large effect on the parallelism of a program. Due to how views are dynamically created and reduced in the associative mechanism, the more successful steals there are, the more work is spent on initialization and reduction of views. The number of successful steals in a program is in turn bounded by the span of the program as analyzed in [18]. This causes a compounding effect — reductions may fall onto the critical path of a program and increase its span, which increases the number of steals that occur, which in turn increases the number of reductions performed and potentially further increases the span of the program. As analyzed in [55], the worst case impact of a reducer with a τ-cost reduction on the effective parallelism of a program can be bounded by a factor of τ^2.

Under the commutative mechanism, the number of reductions performed is fixed based on the number of `reducer_merge` calls and the number of workers. It does not depend on how the program is scheduled on these workers. In many common use cases, `reducer_merge` is only called once after a long period of parallel accumulation. In such programs, reduction overheads are negligible and have nearly no impact on a program's work or parallelism.

9.5 Microbenchmarks

This section compares the performance of the associative and commutative mechanisms in practice. Experiments were performed on an AWS EC2 c4.8xlarge instance, which uses the 2.9 GHz Intel Xeon E5-2666 v3 Processor with 18 cores and has 60 GiB DRAM. To reduce performance anomalies, benchmarks are executed using the

tools `taskset -c 0-n` and `numactl -i all`. Results shown are the median running time of 20 runs.

We start with a simple sum benchmark, which involves taking the sum of the elements of an input array containing random elements. This benchmark can be parallelized with a simple sum reducer, as shown in Figure 9-5.

```
void sum_zero(long* s) {
  *s = 0;
}
void sum_add(long* ls, long* rs) {
  *ls += *rs
}

int parallel_sum(uint8_t* array, int len) {
  long __reducer__((sum_add, sum_zero)) n;
  cilk_for (int i = 0; i < len; i++) {
    n += array[i];
  }
  reducer_merge(n);  // Included only for the commutative mechanism
  return n;
}
```

Figure 9-5: The parallel sum microbenchmark, using simplified reducer linguistics.

A few details regarding this benchmark are worth pointing out:

- An initial version of the benchmark exhibited high performance variance for particular combinations of input array size and worker number. After some investigation, the likely cause is determined to be the fact that a view of the reducer, type `long`, is vulnerable to false sharing. This problem is fixed by padding the reducer type to a cache line in size.

- `cilk_for` is implemented using `cilk_spawn` and `cilk_sync` in a recursive binary divide-and-conquer fashion. The base case, a serial `for` loop, occurs once the number of iterations covered is under some threshold called the ***grain size***. While using a grain size larger than 1 generally improves performance by reducing overheads, a grain size of 1 is used in this benchmark as the goal is to compare the two mechanisms rather than to maximize performance.

- The amount of time taken on a single run of `sum` depends on the size of the input array. To make it easier to compare performance trends across input arrays of different sizes, the `sum` benchmark is run serially for a number of repetitions, so that the total amount of work taken is the same. To be precise, `sum` is run for 2^{24-x} repetitions when the input array has size 2^x.

Timing results are shown in Figure 9-6. As can be seen, the associative mechanism's performance on the `sum` benchmark ranges from 1% to 30% slower than the commutative mechanism, depending on the size of the input array and the number of workers.

This performance difference is greater when the parallelism of the program is low relative to the number of workers, or, in other words, when there is little to no additional benefit gained from adding more workers. This behavior is not unexpected. In a program with low parallelism and ample opportunities for steals — as is the case for a simple `cilk_for` loop — a larger number of successful steals tend to occur. As a result, the associative mechanism needs to spend more work creating and reducing views, resulting in a worse performance.

To better understand whether the additional complexity of supporting commutative reducers is justified, we investigate how much benefit the commutative mechanism can offer in the most extreme case. A microbenchmark maximizes the comparative advantage of commutative reducers if it uses a reducer with an expensive reduction of cost τ, has relatively low parallelism, and offers ample opportunities for steals. In such a program, we may expect to see a significant performance problem with the associative mechanism due to the τ^2 factor impact on the parallelism of a program, as analyzed in [55].

The `histogram` microbenchmark offers a natural example of such behavior. This benchmark involves counting the number of occurrences of each value in an input array of integers, producing a histogram of the array's distribution of values. A serial version of this benchmark is shown in Figure 9-7.

Parallelizing this program using a reducer requires a method of combining views of the histogram. Reducing together two histograms involves adding together the

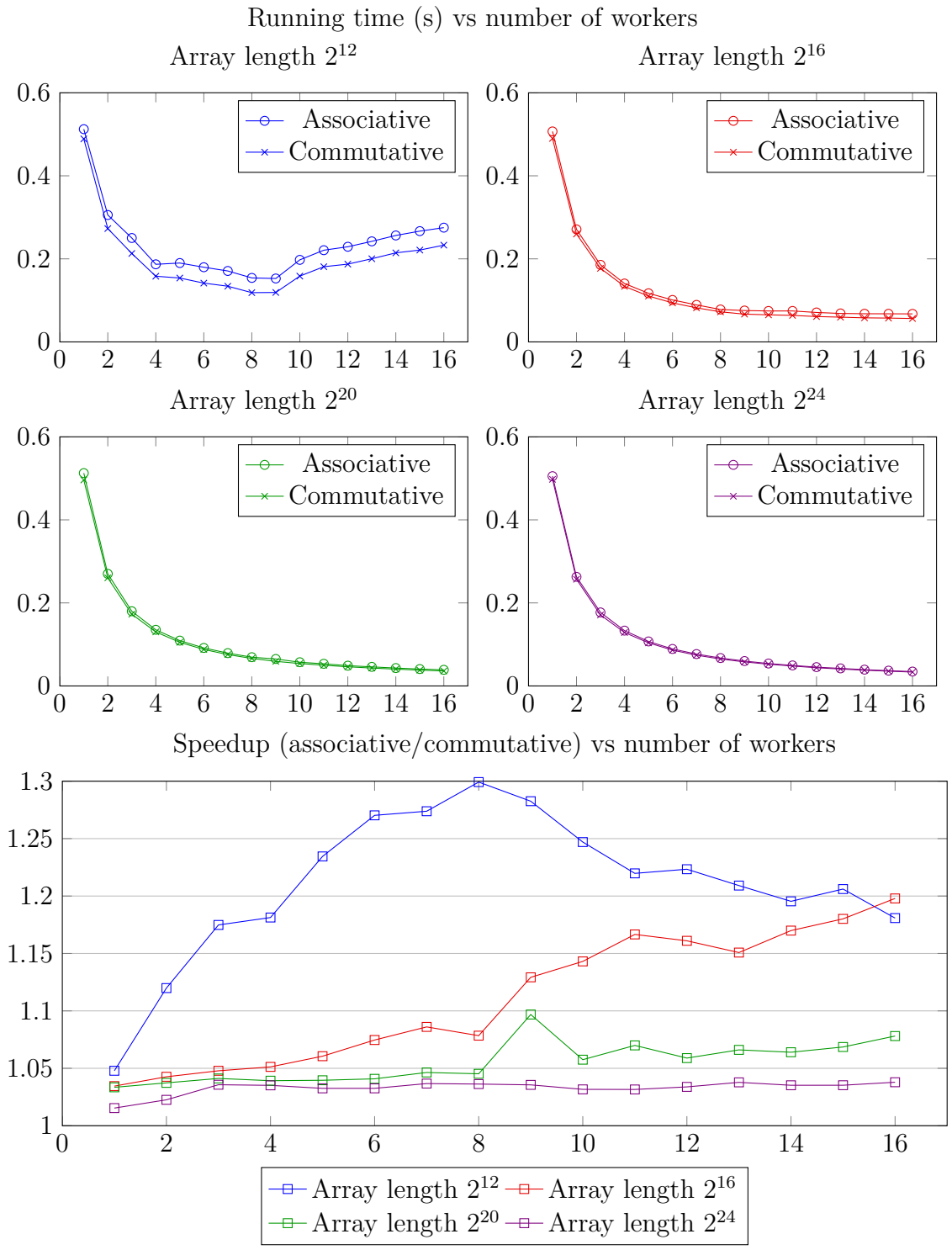

Figure 9-6: Results of running the `parallel_sum` benchmark on input arrays of varying sizes.

```
typedef struct Hist {
  int counts[MAX_VAL];
} Hist;

// Computes a histogram of how often each value appears
Hist histogram(val_t* array, int len) {
  Hist hist = {};
  for (int i = 0; i < len; i++)
    hist.counts[array[i]]++;
  return hist;
}
```

Figure 9-7: The serial histogram microbenchmark.

number of of occurrences of each possible value, or, in other words, adding together the two `count` arrays term-by-term. A parallel version of this benchmark is shown in Figure 9-8.

```
void hist_zero(Hist* hist) {
  for (int i = 0; i < MAX_VAL; i++)
    hist->counts[i] = 0;
}
void hist_add(Hist* lhist, Hist* rhist) {
  for (int i = 0; i < MAX_VAL; i++)
    lhist->counts[i] = lhist->counts[i] + rhist->counts[i];
}

Hist parallel_histogram(val_t* array, int len) {
  Hist __reducer__((hist_add, hist_zero)) hist;
  cilk_for (int i = 0; i < len; i++)
    hist.counts[array[i]]++;
  reducer_merge(hist);  // Included only for the commutative mechanism
  return hist;
}
```

Figure 9-8: The parallel histogram microbenchmark, using simplified reducer linguistics.

A reduction operation of the histogram reducer is expensive, as it requires summing over all entries of an entire array. This behavior meets our goal. To further increase the cost of the reduction operation, vectorization is disabled using the flags

`-fno-vectorize` and `-fno-slp-vectorize`.

The exact cost of the reduction depends on the size of the histogram. We examine two versions of this benchmark, one where the entries of the input array are of type `uint8_t` and the histogram has a length of $2^8 = 256$, one where the entries of the input array are of type `uint16_t` and the histogram has a length of $2^{16} = 65536$.

Timing results are shown in Figures 9-9 and 9-10.

As can be seen in Figure 9-9, the behavior of the histogram benchmark on input arrays of type `uint8_t` is similar to the `parallel_sum` benchmark. The associative mechanism's performance ranges from 2% to 30% slower than the commutative mechanism, and the performance difference is more significant when the input array is smaller and the program has less parallelism. A reduction operation that pairwise sums two 256-element arrays appears to not have a high enough cost to majorly impact the comparative performance of the associative and commutative reducer mechanisms.

In contrast, Figure 9-10 displays very different trends. The histogram benchmark on input arrays of type `uint16_t` shows much greater performance differences between the two mechanisms. The benchmarks on smaller input arrays of size 2^{12} and 2^{16} are not shown, as the cost of reductions overwhelms the amount of useful work. On larger arrays of size 2^{20} and 2^{24}, the commutative mechanism shows a clear advantage over the associative mechanism. In the most extreme case shown here, given an input array of size 2^{20} and on 16 workers, the commutative mechanism performs over $3\times$ faster than the associative. These performance differences are unlikely to be seen in real use cases, however; normal reductions do not take tens of thousands of instructions to execute.

These microbenchmarks show that the commutative mechanism offers a moderate performance advantage over the existing associative mechanism in realistic use cases, and that the difference is more significant in programs with low parallelism. It is uncertain whether this performance improvement justifies the additional complexity of supporting the new commutative mechanism in Cilk, especially given its weaker semantic properties.

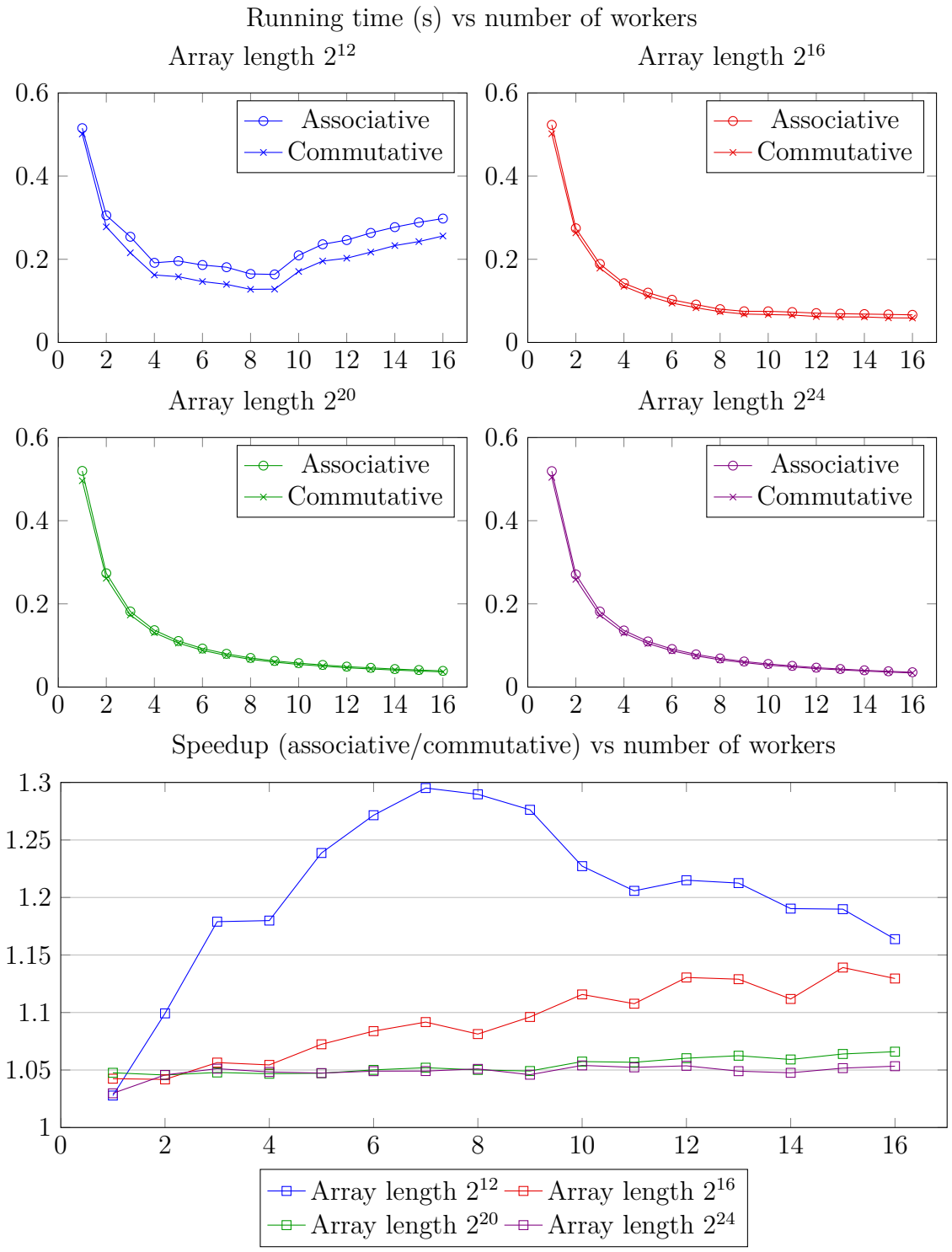

Figure 9-9: Results of running the `parallel_histogram` benchmark with `uint8_t` type input arrays of varying sizes.

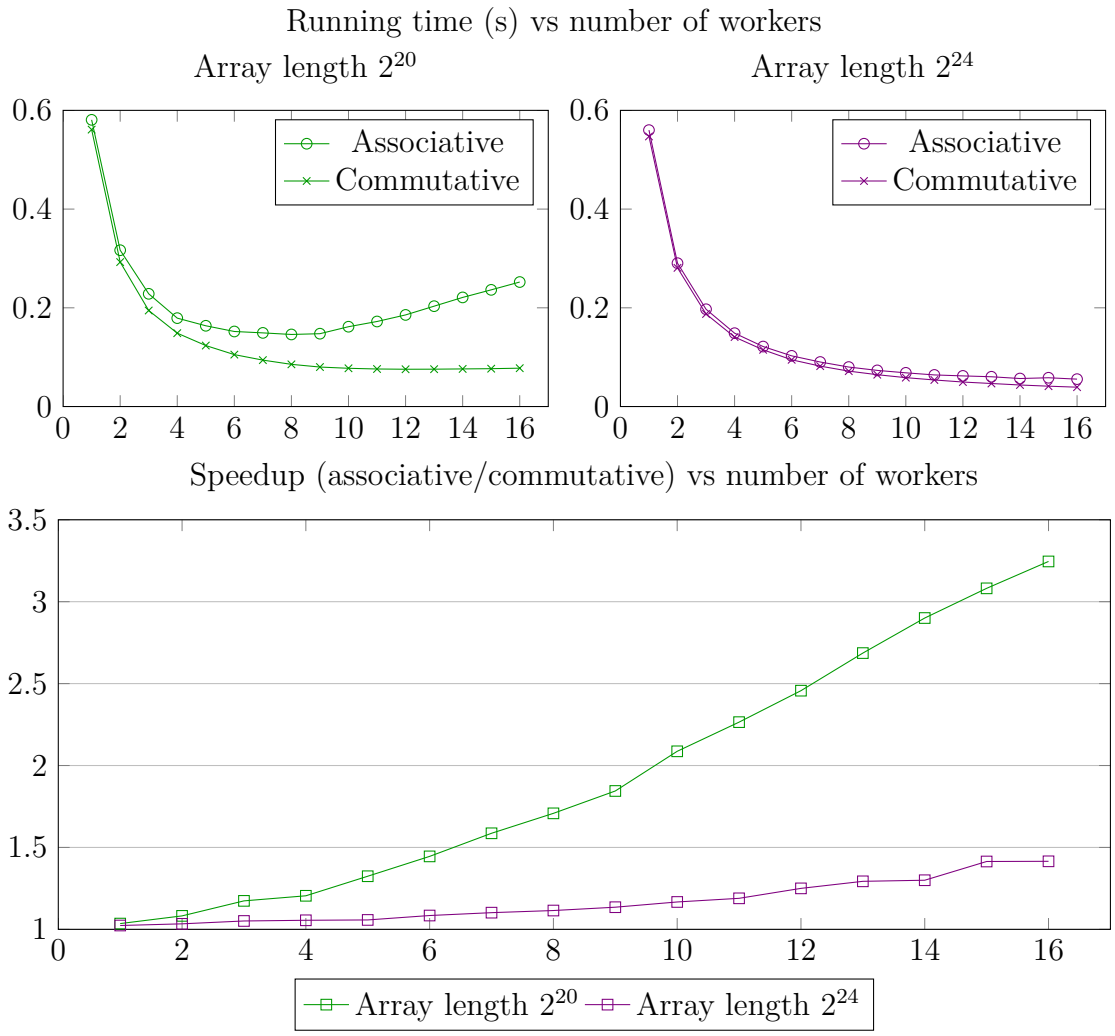

Figure 9-10: Results of running the `parallel_histogram` benchmark with `uint16_t` type input arrays of varying sizes.

Chapter 10

The Multiplicative Version of Azuma's Inequality, with an Application to Contention Analysis in Work Stealing Schedulers

Azuma's inequality is a tool for proving concentration bounds on random variables. The inequality can be thought of as a natural generalization of additive Chernoff bounds. On the other hand, the analogous generalization of multiplicative Chernoff bounds has, to our knowledge, never been explicitly formulated.

We formulate a multiplicative-error version of Azuma's inequality. We then show how to apply this new inequality in order to greatly simplify (and correct) the analysis of contention delays in multithreaded systems managed by randomized work stealing.

10.1 Introduction

One of the most widely used tools in algorithm analysis is the Chernoff bound, which gives a concentration inequality on sums of independent random variables. The Cher-

noff bound exists in many forms, but the two most common variants are the additive and multiplicative bounds:

Theorem 28 (Additive Chernoff Bound)**.** Let $X_1, \ldots, X_n \in \{0, 1\}$ be independent random variables, and let $X = \sum_{i=1}^{n} X_i$. Then for any $\varepsilon > 0$,

$$\mathbb{P}\left[X \geq \mathbb{E}[X] + \varepsilon\right] \leq \exp\left(-\frac{2\varepsilon^2}{n}\right).$$

Theorem 29 (Multiplicative Chernoff Bound)**.** Let $X_1, \ldots, X_n \in \{0, 1\}$ be independent random variables. Let $X = \sum_{i=1}^{n} X_i$ and let $\mu = \mathbb{E}[X]$. Then for any $\delta > 0$,

$$\mathbb{P}[X \geq (1+\delta)\mu] \leq \exp\left(-\frac{\delta^2 \mu}{2+\delta}\right)$$

and for any $0 < \delta < 1$,

$$\mathbb{P}[X \leq (1-\delta)\mu] \leq \exp\left(-\frac{\delta^2 \mu}{2}\right).$$

Although the additive Chernoff bound is often convenient to use, the multiplicative bound can in some cases be much stronger. Suppose, for example, that $X_1, X_2, \ldots, X_n$ each take value 1 with probability $(\log n)/n$. By the additive bound, one can conclude that $\sum_i X_i = O(\sqrt{n \log n})$ with high probability in n. On the other hand, the multiplicative bound can be used to show that $\sum_i X_i = O(\log n)$ with high probability in n. In general, whenever $\mathbb{E}[X] \ll n$, the multiplicative bound is more powerful.

Handling dependencies with Azuma's inequality. Chernoff bounds require that the random variables $X_1, X_2, \ldots, X_n$ be independent. In many algorithmic applications, however, the X_i's are not independent, such as when analyzing algorithms in which $X_1, X_2, \ldots, X_n$ are the results of decisions made by an ***adaptive*** adversary over time. When analyzing these applications (see, e.g., [2,3,5,8,9,19,23,28,33,43,44, 51,52,57,74]), a stronger inequality known as ***Azuma's inequality*** is often useful.

Theorem 30 (Azuma's inequality)**.** Let $Z_0, Z_1, \ldots, Z_n$ be a supermartingale, meaning that $\mathbb{E}[Z_i \mid Z_0, \ldots, Z_{i-1}] \leq Z_{i-1}$. Assume additionally that $|Z_i - Z_{i-1}| \leq c_i$. Then

for any $\varepsilon > 0$,

$$\mathbb{P}[Z_n - Z_0 \geq \varepsilon] \leq \exp\left(-\frac{\varepsilon^2}{2\sum_{i=1}^n c_i^2}\right).$$

By applying Azuma's inequality to the exposure martingale for a sum $\sum_i X_i$ of random variables, one arrives at the following corollary, which is often useful in analyzing randomized algorithms (for direct applications of Corollary 31, see, e.g., [21–23, 35, 51, 57]).

Corollary 31. Let $X_1, X_2, \ldots, X_n$ be random variables satisfying $X_i \in [0, c_i]$. Suppose that $\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}] \leq p_i$ for all i. Then for any $\varepsilon > 0$,

$$\mathbb{P}\left[\sum_i X_i \geq \sum_i p_i + \varepsilon\right] \leq \exp\left(-\frac{\varepsilon^2}{2\sum_{i=1}^n c_i^2}\right).$$

This work: an inequality with multiplicative error. Azuma's inequality can be viewed as a generaliation of ***additive*** Chernoff bounds. In this work, we formulate the ***multiplicative*** analog to Azuma's inequality.

Theorem 32. Let $Z_0, Z_1, \ldots, Z_n$ be a supermartingale, meaning that $\mathbb{E}[Z_i \mid Z_0, \ldots, Z_{i-1}] \leq Z_{i-1}$. Assume additionally that $-a_i \leq Z_i - Z_{i-1} \leq b_i$, where $a_i + b_i = c$ for some constant $c > 0$ independent of i. Let $\mu = \sum_{i=1}^n a_i$. Then for any $\delta > 0$,

$$\mathbb{P}[Z_n - Z_0 \geq \delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

This theorem yields the following corollary.

Corollary 33. Let $X_1, \ldots, X_n \in [0, c]$ be real-valued random variables with $c > 0$. Suppose that $\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}] \leq a_i$ for all i. Let $\mu = \sum_{i=1}^n a_i$. Then for any $\delta > 0$,

$$\mathbb{P}\left[\sum_i X_i \geq (1+\delta)\mu\right] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

In the same way that multiplicative Chernoff bounds are in some cases much stronger than additive Chernoff bounds, the multiplicative Azuma's inequality is in some cases much stronger than the standard (additive) Azuma's inequality, as occurs, in particular, when $\sum_i a_i \ll cn$.

As far as we know, no one to date has explicitly formulated the multiplicative version of Azuma's inequality. Our work is targeted towards algorithm designers. Our hope is that Theorem 32 will simplify the task of analyzing randomized algorithms, providing an instrument that can be used in place of custom Chernoff bounds and ad-hoc combinatorial arguments.

Extensions. We present two extensions of Theorem 32 and Corollary 33.

In Section 10.4, we generalize Theorem 32 so that $a_1, a_2, \ldots, a_n$ are determined by an ***adaptive adversary***. This means that each a_i can be partially a function of $Z_0, \ldots, Z_{i-1}$. As long as the a_i's are restricted to satisfy $\sum_i a_i \leq \mu$, then the bound from Theorem 32 continues to hold. We also discuss several applications of the adaptive version of the theorem.

In Appendix B, we extend Theorem 32 to give a lower tail bound. In particular, just as Theorem 32 gives an upper tail bound for supermartingales, a similar approach gives a lower tail bound for submartingales (also with multiplicative error).

An application: work stealing. In order to demonstrate the power of Theorem 32 we revisit a classic result in multithreaded scheduling. In the problem of scheduling multithreaded computations on parallel computers, a fundamental question is how to decide when one processor should "steal" computational threads from another. In the seminal paper, ***Scheduling Multithreaded Computations by Work Stealing*** [18], Blumofe and Leiserson presented the first provably good work-stealing scheduler for multithreaded computations with dependencies. The paper has been influential to both theory and practice, amassing almost two thousand citations, and inspiring the Cilk Programming language and runtime system [17, 42, 67].

One result in [18] is an analysis of the so-called (P, M)-recycling game,[1] which models the contention incurred by a randomized work-stealing algorithm. By combining the analysis of the (P, M)-recycling game with a delay-sequence argument, the authors are able to bound the execution time and communication cost of their randomized work-stealing algorithm.

The (P, M)-recycling game takes place on P bins which are initially empty. In each step of the game, if there are k balls presently in the bins, then the player selects some value $j \in \{0, 1, \ldots, P - k\}$ and then tosses j balls at random into bins. At the end of each step, one ball is removed from each non-empty bin. The game continues until M total tosses have been made. The goal of the player is to maximize the ***total delay*** experienced by the balls, where the delay of a ball b thrown into a bin i is defined to be the number of balls already present in bin i at the time of b's throw. Lemma 6 of [18] states that, even if the player is an adaptive adversary, the total delay is guaranteed to be at most $O(M + P \log P + P \log \epsilon^{-1})$ with probability at least $1 - \epsilon$.

In part due to lack of good analytical tools, the authors of [18] attempt to analyze the (P, M)-recycling game via a combinatorial argument. Unfortunately, the argument fails to notice certain subtle (but important) dependencies between random variables, and consequently the analysis is incorrect.[2]

In Section 10.3, we give a simple and short analysis of the (P, M)-recycling using Theorem 32. We also explain why the same argument does not follow from the standard Azuma's inequality. In addition to being simpler (and more correct) than the analysis in [18], our analysis enables the slightly stronger bound of $O(M + P \log \epsilon^{-1})$.

Related Work

Although Chernoff bounds are often attributed to Herman Chernoff, they were originally formulated by Herman Rubin (see discussion in [24]). Azuma's inequality, on

[1] Not to be confused with the ball recycling game of [7].

[2] We thank Charles Leiserson of MIT, one of the original authors of [18], for suggesting that the analysis in [18] should be revisited.

the other hand, was independently formulated by several different authors, including Kazuoki Azuma [6], Wassily Hoeffding [46], and Sergei Bernstein [14] (although in a slightly different form). As a result, the inequality is sometimes also referred to as the Azuma-Hoeffding inequality.

The key technique used to prove Azuma's inequality is to apply Markov's inequality to the moment generating function of a random variable. This technique is well understood and has served as the foundation for much of the work on concentration inequalities in statistics and probability theory [12, 30, 36, 37, 40, 45, 48, 56, 58–60, 63–65, 72] (see [20] or [25] for a survey). Extensive work has been devoted to generalizing Azuma's inequality in various ways. For example, Bernstein-type inequalities parameterize the concentration bound by the kth moments of the random variables being summed [12–14, 30, 36, 37, 40, 45, 48, 63]. Most of the research in this direction has been targeted towards applications in statistics and probability theory, rather than to theoretical computer science.

The main contribution of this work is to explicitly formulate the multiplicative analogue of Azuma's inequality, and to discuss its application within algorithm analysis. We emphasise that the proof of the inequality is not, in itself, a substantial contribution, since the inequality is relatively straightforward to derive by combining the proof of the multiplicative Chernoff bound with that of Azuma's inequality. Nonetheless, by presenting the theorem as a tool that can be directly referenced by algorithm designers, we hope to simplify the task of proving concentration bounds within the context of algorithm analysis.

Besides Azuma's inequality, there are several other generalizations of Chernoff bounds that are used in algorithm analysis. Chernoff-style bounds have been shown to apply to sums of random variables that are negatively associated, rather than independent, and several works have developed useful techniques for identifying when random variables are negatively associated [34, 47, 49, 73]. Another common approach is to show that a sum X of not necessarily independent random variables is stochastically dominated by a sum X' of independent random variables (see Lemma 3 of [4]), thereby allowing for the application of Chernoff bounds to X.

10.2 Multiplicative Azuma's Inequality

In this section we prove the following theorem and corollary.

Theorem 32. Let $Z_0, Z_1, \ldots, Z_n$ be a supermartingale, meaning that $\mathbb{E}[Z_i \mid Z_0, \ldots, Z_{i-1}] \leq Z_{i-1}$. Assume additionally that $-a_i \leq Z_i - Z_{i-1} \leq b_i$, where $a_i + b_i = c$ for some constant $c > 0$ independent of i. Let $\mu = \sum_{i=1}^{n} a_i$. Then for any $\delta > 0$,

$$\mathbb{P}[Z_n - Z_0 \geq \delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

Corollary 33. Let $X_1, \ldots, X_n \in [0, c]$ be real-valued random variables with $c > 0$. Suppose that $\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}] \leq a_i$ for all i. Let $\mu = \sum_{i=1}^{n} a_i$. Then for any $\delta > 0$,

$$\mathbb{P}\left[\sum_i X_i \geq (1+\delta)\mu\right] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

We start our proof by establishing a simple inequality.

Lemma 34. For any $t > 0$ and any random variable X such that $\mathbb{E}[X] \leq 0$ and $-a \leq X \leq b$,

$$\mathbb{E}\left[e^{tX}\right] \leq \exp\left(\frac{a}{a+b}\left(e^{t(a+b)} - 1\right) - ta\right).$$

Proof. Consider the linear function f defined on $[-a, b]$ that passes through points $(-a, e^{-ta})$ and (b, e^{tb}). Since e^{tx} is convex, Jensen's inequality states that f upper bounds e^{tx}, implying that $\mathbb{E}[e^{tX}] \leq \mathbb{E}[f(X)]$. Since f is linear, $\mathbb{E}[f(X)]$ only depends on $\mathbb{E}[X]$, and one can derive that

$$\mathbb{E}[f(X)] = \frac{b - \mathbb{E}[X]}{a+b}e^{-ta} + \frac{a + \mathbb{E}[X]}{a+b}e^{tb}.$$

This quantity is maximized when $\mathbb{E}[X]$ is maximized at $\mathbb{E}[X] = 0$. Therefore,

$$\mathbb{E}[e^{tX}] \leq \frac{b}{a+b}e^{-ta} + \frac{a}{a+b}e^{tb}$$

$$= e^{-ta}\left(1 + \frac{a}{a+b}\left(e^{t(a+b)} - 1\right)\right)$$
$$\leq \exp\left(\frac{a}{a+b}\left(e^{t(a+b)} - 1\right) - ta\right).$$

□

Proof of Theorem 32. By Markov's inequality, for any $t > 0$ and v,

$$\mathbb{P}[Z_n - Z_0 \geq v] = \mathbb{P}\left[e^{t(Z_n - Z_0)} \geq e^{tv}\right]$$
$$\leq \frac{\mathbb{E}[e^{t(Z_n - Z_0)}]}{e^{tv}}. \tag{10.1}$$

Let $X_i = Z_i - Z_{i-1}$. Since Z_i is a supermartingale, for any i, $\mathbb{E}[X_i \mid Z_0, \ldots, Z_{i-1}] \leq 0$. Moreover, from the assumptions in the problem, $-a_i \leq X_i \leq b_i$. Therefore, Lemma 34 applies to $X = (X_i \mid Z_0, \ldots, Z_{i-1})$, and we have

$$\mathbb{E}[e^{tX_i} \mid Z_0, \ldots, Z_{i-1}] \leq \exp\left(\frac{a_i}{c}\left(e^{tc} - 1\right) - ta_i\right). \tag{10.2}$$

In the following derivation, which will involve expectations of expectations, it will be important to understand which random variables each expectation is taken over. We will adopt the notation $\mathbb{E}_S[f(S)]$ to denote an expectation taken over a set of random variables S. Using (10.2) along with the law of total expectation, which states that $\mathbb{E}_{X,Y}[A] = \mathbb{E}_X[\mathbb{E}_Y[A|X]]$ for any random variable A that is a function of random variables X and Y, we derive

$$\mathbb{E}_{Z_0, X_1, \ldots, X_{i-1}, X_i}\left[\prod_{j=1}^{i} e^{tX_j}\right] = \mathbb{E}_{Z_0, X_1, \ldots, X_{i-1}}\left[\mathbb{E}_{X_i}\left[\prod_{j=1}^{i} e^{tX_j} \,\middle|\, Z_0, X_1, \ldots, X_{i-1}\right]\right]$$
$$= \mathbb{E}_{Z_0, X_1, \ldots X_{i-1}}\left[\left(\prod_{j=1}^{i-1} e^{tX_j}\right)\mathbb{E}_{X_i}\left[e^{X_i} \,\middle|\, Z_0, X_1 \ldots, X_{i-1}\right]\right]$$
$$= \mathbb{E}_{Z_0, X_1, \ldots X_{i-1}}\left[\left(\prod_{j=1}^{i-1} e^{tX_j}\right)\mathbb{E}_{X_i}\left[e^{X_i} \,\middle|\, Z_0, Z_1, \ldots, Z_{i-1}\right]\right]$$
$$\leq \mathbb{E}_{Z_0, X_1, \ldots X_{i-1}}\left[\left(\prod_{j=1}^{i-1} e^{tX_j}\right)\exp\left(\frac{a_i}{c}\left(e^{tc} - 1\right) - ta_i\right)\right]$$

$$= \exp\left(\frac{a_i}{c}\left(e^{tc}-1\right) - ta_i\right) \mathbb{E}_{Z_0,X_1,\ldots X_{i-1}}\left[\prod_{j=1}^{i-1} e^{tX_j}\right].$$

By applying the above inequality iteratively, we arrive at the following:

$$\begin{aligned}
\mathbb{E}[e^{t(Z_n - Z_0)}] &= \mathbb{E}_{Z_0,X_1,\ldots,X_n}\left[\prod_{i=1}^{n} e^{tX_i}\right] \\
&\leq \prod_{i=1}^{n} \exp\left(\frac{a_i}{c}(e^{tc}-1) - ta_i\right) \\
&= \exp\left(\frac{\mu}{c}\left(e^{tc}-1\right) - t\mu\right).
\end{aligned}$$

By (10.1), we have

$$\mathbb{P}[Z_n - Z_0 \geq v] \leq \exp\left(\frac{\mu}{c}\left(e^{tc}-1\right) - t\mu - tv\right).$$

Plugging in $t = (\ln(1+\delta))/c$ and $v = \delta\mu$ for $\delta > 0$ yields

$$\begin{aligned}
\mathbb{P}[Z_n - Z_0 \geq \delta\mu] &\leq \exp\left(\frac{\mu\delta}{c} - \frac{\mu}{c}\ln(1+\delta) - \frac{\mu}{c}\delta\ln(1+\delta)\right) \\
&= \exp\left(\frac{\mu}{c}\left(\delta - (1+\delta)\ln(1+\delta)\right)\right).
\end{aligned}$$

For any $\delta > 0$,

$$\delta - (1+\delta)\ln(1+\delta) \leq -\frac{\delta^2}{2+\delta},$$

which can be seen by inspecting the derivative of both sides.[3] As a result,

$$\mathbb{P}[Z_n - Z_0 \geq \delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

□

Remark. A stronger but more unwieldy bound may sometimes be helpful. By skip-

[3]Consider $f(x) = x/(1+x) - \ln(1+x) + x^2/((1+x)(2+x))$. Then $f(0) = 0$ and $f'(x) = -x^2/\left((1+x)(2+x)^2\right) \leq 0$ for $x \geq 0$. Therefore, $f(x) \leq 0$ for $x \geq 0$, and the inequality holds for $\delta > 0$.

ping the approximation of $\delta - (1+\delta)\ln(1+\delta)$, we derive

$$\mathbb{P}[Z_n - Z_0 \geq \delta\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\mu/c}.$$

We conclude the section by proving Corollary 33.

Proof of Corollary 33. Define $Z_i = \sum_{j=1}^{i}(X_j - a_j)$. Note that $Z_i - Z_{i-1} = X_i - a_i$. The given condition

$$\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}] \leq a_i$$

implies that

$$\mathbb{E}[Z_i - Z_{i-1} \mid Z_0, \ldots, Z_{i-1}] = \mathbb{E}[Z_i - Z_{i-1} \mid X_1, \ldots, X_{i-1}] \leq 0$$

and thus that Z_i is a supermartingale. Moreover, as each $X_i \in [0, c]$, we have that $Z_i - Z_{i-1} \geq -a_i$, $Z_i - Z_{i-1} \geq c - a_i$. Setting $\mu = \sum_{i=1}^{n} a_i$, Theorem 32 implies

$$\mathbb{P}[Z_n - Z_0 \geq \delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

We may break down $Z_n - Z_0$ as

$$\begin{aligned} Z_n - Z_0 &= \sum_{i=1}^{n}(Z_i - Z_{i-1}) \\ &= \sum_{i=1}^{n}(X_i - a_i) \\ &= \sum_{i=1}^{n} X_i - \mu. \end{aligned}$$

Therefore,

$$\mathbb{P}\left[\sum_{i=1}^{n} X_i \geq (1+\delta)\mu\right] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

□

10.3 Analyzing the (P, M)-Recyling Game

In this section we revisit the analysis of the (P, M)-recycling game given in [18]. We begin by defining the game and explaining why the analysis given in [18] is incorrect. Then we apply Theorem 32 to obtain a simple and correct analysis.

Defining the Game

The (P, M)-recycling game is a combinatorial game, in which balls labelled 1 to P are tossed at random into P bins. Initially, all P balls are in a reservoir separate from the P bins. At each step of the game, the player executes the following two operations in sequence:

1. The player chooses some of the balls in the reservoir (possibly all and possibly none). For each of these balls, the player removes it from the reservoir, selects one of the P bins uniformly and independently at random, and tosses the ball into it.

2. The player inspects each of the P bins in turn, and for each bin that contains at least one ball, the player removes any one of the balls in the bin and returns it to the reservoir.

The player is permitted to make a total of M ball tosses. The game ends when M ball tosses have been made and all balls have been removed from the bins and placed back in the reservoir. The player is allowed to base their strategy (how many/which balls to toss) depending on outcomes from previous turns.

After each step t of the game, there are some number n_t of balls left in the bins. The ***total delay*** is defined as $D = \sum_{t=1}^{T} n_t$, where T is the total number of steps in the game. Equivalently, if we define the ***delay*** of a ball b being tossed into a bin i to be the number of balls already present in bin i at the time of the toss, then the total delay is the sum of the delays of all ball tosses.

We would like to give high probability bounds on the total delay, no matter what strategy the player takes.

An Incorrect Analysis of the Recycling Game

The following bound is given by [18].

Theorem 35 (Lemma 6 in [18])**.** For any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the total delay in the (P, M)-recycling game is $O(M + P \log P + P \log \varepsilon^{-1})$.

In order to prove Theorem 35, the authors [18] sketch a complicated combinatorial analysis of the game. Define the indicator random variable x_{ir} to be 1 if the ith toss of ball 1 is delayed by ball r, and 0 otherwise. A key component in the analysis [18] is to show that,[4] for any set $R \subseteq [P]$ of balls,

$$\Pr[x_{ir} \text{ for all } r \in R] \leq P^{-|R|}. \tag{10.3}$$

Unfortunately, due to subtle dependencies between the random variables x_{ir}, (10.3) is not true (or even close to true). To see why, suppose that the player (i.e., the adversary) takes the following strategy: Throw balls $A = \{2, 3, \ldots, P\}$ in the first step. If the balls in A do not land in the same bin, then wait $P - 1$ steps until all bins are empty, and throw the balls in A again. Continue rethrowing until there is some step t in which all of the balls in A land in the same bin. At the end of step t, remove ball 2, leaving balls $3, 4, \ldots, P$ in the same bin as each other. Then on step $t + 1$ perform the first throw of ball 1.

If M is sufficiently large so that all balls in A almost certainly land together before the process ends, then the probability that the first throw of ball 1 lands in the same bin as balls $3, 4, \ldots, P$ is approximately $1/P$. In contrast, (10.3) claims to bound the same probability by $1/P^{P-2}$.

The difficulty of proving Theorem 35 via an ad-hoc combinatorial argument is further demonstrated by another error in [18]'s analysis. Throughout the proof, the authors define m_i to be the number of times that ball i is thrown, and then treat each m_i as taking a fixed value. In actuality, however, the m_i's are random variables that are partially controlled by an adversary (i.e., the player of the game), meaning

[4] In fact, the analysis requires a somewhat stronger property to be shown. But for simplicity of exposition, we focus on this simpler variant.

that the outcomes of the m_i's may be linked to the outcomes of the $x_{i,r}$'s. This consideration adds even further dependencies that must be considered in order to obtain a correct analysis.

A Simple and Correct Analysis Using Multiplicative Azuma's Inequality

We now give a simple (and correct) analysis of the (P, M)-recycling game using the multiplicative version of Azuma's inequality. In fact, we prove a slightly stronger bound than Theorem 35.

Theorem 36. For any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the total delay in the (P, M)-recycling game is $O(M + P \log(1/\varepsilon))$.

Proof. For $i = 1, 2, \ldots, M$, define the delay X_i of the ith toss to be the number of balls in the bin that the ith toss lands in, not counting the ith toss itself. The total delay can be expressed as $D = \sum_{i=1}^{M} X_i$.

As the player's strategy can adapt to the outcomes of previous tosses, the X_i's may have complicated dependencies. Nonetheless, since there are at most $P-1$ balls present at time of the ith toss, we know that $X_i \in [0, P]$. Moreover, since the toss selects a bin $\{1, 2, \ldots, P\}$ at random, each ball present at the time of the toss has probability $1/P$ of contributing to the delay X_i. Thus, no matter the outcomes of $X_1, \ldots, X_{i-1}$, we have that $\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}] \leq (P-1)/P \leq 1$. We can therefore apply Corollary 33, with $a_i = 1$ for all i and $c = P$, to deduce that

$$\Pr[D \geq (1+\delta)M] \leq \exp\left(-\frac{\delta^2 M}{(2+\delta)P}\right). \tag{10.4}$$

If $M \geq P \ln(1/\varepsilon)$, we may substitute $\delta = 2$ into (10.4) to derive $\mathbb{P}[D \geq 3M] \leq \exp(-M/P) \leq \varepsilon$. If $M \leq P \ln(1/\varepsilon)$, we may instead substitute $\delta = 2P \ln(1/\varepsilon)/M$. As $\delta \geq 2$, we have $\delta/(2+\delta) \geq 1/2$, and we derive $\mathbb{P}\left[D \geq M + 2P \ln(1/\varepsilon)\right] \leq \exp(-\delta M/(2P)) = \varepsilon$.

In either case, $\mathbb{P}[D \geq 3M + 2P\ln(1/\varepsilon)] \leq \varepsilon$, which proves the theorem statement. □

Why Standard Azuma's Inequality Does Not Suffice

In order to fully understand the proof of Theorem 36, it is informative to consider what happens if we attempt to use (the standard) Azuma's inequality to analyze $D = \sum_{i=1}^{M} X_i$. Applying Corollary 31 with $c_i = P$ for all i, we get that

$$\Pr[D > (1+\delta)M] \leq \exp\left(-\frac{(\delta M)^2}{2MP^2}\right) = \exp\left(-\frac{\delta^2 M}{2P^2}\right). \tag{10.5}$$

In contrast, for $\delta \geq 2$, Corollary 33 gives a bound of

$$\Pr[D > (1+\delta)M] \leq \exp\left(-\frac{\delta^2 M}{(2+\delta)P}\right) \leq \exp\left(-\frac{\delta M}{2P}\right). \tag{10.6}$$

Since $D \leq PM$ trivially, the interesting values for δ are $\delta \leq P$. On the other hand, for all δ satisfying $2 \leq \delta < P$, the bound given by (10.6) is stronger than the bound given by (10.5). The reason that the multiplicative version of Azuma's does better than the additive version is that the random variables X_i have quite small means, meaning that the a_i's used by the multiplicative bound are much smaller than the c_i's used by the additive bound. When δ is a constant, this results in a full factor-of-$\Theta(P)$ difference in the exponent achieved by the two bounds. It is not possible to derive a $O(M + P\log\varepsilon^{-1})$ high probability bound with (10.5) alone.

10.4 Adversarial Multiplicative Azuma's Inequality

In this section, we extend Theorem 32 and Corollary 33 in order to allow for the values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ to be random variables that are determined adaptively. Formally, we define the supermartingale $Z_0, \ldots, Z_n$ with respect to a filtration, and then defining $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ to be predictable processes with respect

to that same filtration.

The statement of Theorem 37 uses several notions that are standard in probability theory (see, e.g., [66] and [15] for formal definitions) but less standard in theoretical computer science.

Theorem 37. Let $Z_0, \ldots Z_n$ be a supermartingale with respect to the filtration $F_0, \ldots, F_n$, and let $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ be predictable processes with respect to the same filtration. Suppose there exist values $c > 0$ and μ, satisfying that $-A_i \leq Z_i - Z_{i-1} \leq B_i$, $A_i + B_i = c$, and $\sum_{i=1}^{n} A_i \leq \mu$ (almost surely). Then for any $\delta > 0$,

$$\mathbb{P}[Z_n - Z_0 \geq \delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

Corollary 38. Suppose that Alice constructs a sequence of random variables $X_1, \ldots X_n$, with $X_i \in [0, c], c > 0$, using the following iterative process. Once the outcomes of $X_1, \ldots, X_{i-1}$ are determined, Alice then selects the probability distribution $\mathcal{D}_i$ from which X_i will be drawn; X_i is then drawn from distribution $\mathcal{D}_i$. Alice is an adaptive adversary in that she can adapt $\mathcal{D}_i$ to the outcomes of $X_1, \ldots, X_{i-1}$. The only constraint on Alice is that $\sum_i \mathbb{E}[X_i \mid \mathcal{D}_i] \leq \mu$, that is, the sum of the means of the probability distributions $\mathcal{D}_1, \ldots, \mathcal{D}_n$ must be at most μ.

If $X = \sum_i X_i$, then for any $\delta > 0$,

$$\mathbb{P}[X \geq (1+\delta)\mu] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

Remark. Formally, a filtration $F_0, \ldots, F_{n-1}$ is a sequence of σ-algebras such that $F_i \subseteq F_{i+1}$ for each i. Informally, one can simply think of the F_i's as revealing "random bits." For each i, F_i reveals the set of random bits used to determine all of $Z_0, \ldots, Z_i$, $A_0, \ldots, A_i$, and $B_0, \ldots, B_i$. The fact that $Z_0, Z_1, \ldots, Z_n$ is a martingale with respect to $F_0, F_1, \ldots, F_{n-1}$ means simply that the random bits F_i determine Z_i (that is, Z_i is F_i-***measurable***), and that $\mathbb{E}[Z_i \mid F_{i-1}] = Z_{i-1}$. The fact that

$A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ are predictable processes, means simply that each A_i and B_i is determined by the random bits F_{i-1} (that is, A_i, B_i are F_{i-1}-***measurable***).

To prove Theorem 37, we prove the following key lemma:

Lemma 39. Let $Z_0, \ldots Z_n$ be a supermartingale with respect to the filtration $F_0, \ldots, F_n$, and let $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ be predictable processes with respect to the same filtration. Suppose there exist values $c > 0$ and μ, satisfying that $-A_i \leq Z_i - Z_{i-1} \leq B_i$, $A_i + B_i = c$, and $\sum_{i=1}^{n} A_i \leq \mu$ (almost surely). Then for any $t > 0$,

$$\mathbb{E}\left[e^{t(Z_n - Z_0)} \;\middle|\; F_0\right] \leq \exp\left(\frac{\mu}{c}(e^{tc} - 1) - t\mu\right).$$

Proof. We proceed by induction on n.

The base case. For $n = 0$, $Z_n - Z_0 = 0$, and for any $c, t > 0$, $(e^{tc} - 1)/c - t > 0$. Therefore, $\mu(e^{tc} - 1)/c - t\mu \geq 0 = t(Z_n - Z_0)$, and the inequality holds.

The inductive step. Assume that this statement is true for $n - 1$, and we shall prove it for n.

The law of total expectation states that for any random variable X and any σ-algebras $H_1 \subseteq H_2$, $\mathbb{E}[\mathbb{E}[X \mid H_2] \mid H_1] = \mathbb{E}[X \mid H_1]$. As $\{F_i\}$ is a filtration, we know $F_{i-1} \subseteq F_i$, and thus

$$\mathbb{E}\left[e^{t(Z_n - Z_0)} \;\middle|\; F_0\right] = \mathbb{E}\left[\mathbb{E}\left[e^{t(Z_n - Z_0)} \;\middle|\; F_1\right] \;\middle|\; F_0\right].$$

Since $e^{t(Z_1 - Z_0)}$ is F_1-measurable, we can pull it out of the expectation as follows:

$$\mathbb{E}\left[\mathbb{E}\left[e^{t(Z_n - Z_0)} \;\middle|\; F_1\right] \;\middle|\; F_0\right] = \mathbb{E}\left[e^{t(Z_1 - Z_0)} \cdot \mathbb{E}\left[e^{t(Z_n - Z_1)} \;\middle|\; F_1\right] \;\middle|\; F_0\right]. \tag{10.7}$$

Let $Z'_i = Z_{i+1}$, $F'_i = F_{i+1}$, $A'_i = A_{i+1}$, $B'_i = B_{i+1}$. We know that $Z'_0, \ldots, Z'_{n-1}$ is a supermartingale with respect to $F'_0, \ldots, F'_{n-1}$. Additionally, we know $A'_1, \ldots, A'_{n-1}$ and $B'_1, \ldots, B'_{n-1}$ are predictable processes with respect to $F'_0, \ldots, F'_{n-1}$ satisfying that $-A'_i \leq Z'_i - Z'_{i-1} \leq B'_i$, $A'_i + B'_i = c$, and $\sum_{i=1}^{n-1} A'_i \leq \mu - (A_1 \mid F_0)$. Therefore,

we may apply our inductive hypothesis to derive

$$\begin{aligned}\mathbb{E}\left[e^{t(Z_n - Z_1)} \;\middle|\; F_1\right] &= \mathbb{E}\left[e^{t(Z'_{n-1} - Z'_0)} \;\middle|\; F'_0\right] \\ &\leq \left(\exp\left(\frac{\mu - A_1}{c}(e^{tc} - 1) - t(\mu - A_1)\right) \;\middle|\; F_0\right). \end{aligned} \tag{10.8}$$

Combining (10.7) and (10.8), we find that

$$\mathbb{E}\left[e^{t(Z_n - Z_0)} \;\middle|\; F_0\right] = \mathbb{E}\left[e^{t(Z_1 - Z_0)} \cdot \exp\left(\frac{\mu - A_1}{c}(e^{tc} - 1) - t(\mu - A_1)\right) \;\middle|\; F_0\right].$$

As A_1 is F_0-measurable, we can pull the exponential term out of the expectation to arrive at

$$\mathbb{E}\left[e^{t(Z_n - Z_0)} \;\middle|\; F_0\right] = \left(\exp\left(\frac{\mu - A_1}{c}(e^{tc} - 1) - t(\mu - A_1)\right) \;\middle|\; F_0\right) \mathbb{E}\left[e^{t(Z_1 - Z_0)} \;\middle|\; F_0\right]. \tag{10.9}$$

Since Z_i is a supermartingale, $\mathbb{E}[Z_1 - Z_0 \mid F_0] \leq 0$. Therefore, Lemma 34 applies to $X = (Z_1 - Z_0 \mid F_0)$, $a = (A_1 \mid F_0)$, $b = (B_1 \mid F_0)$, and we have

$$\mathbb{E}[e^{t(Z_1 - Z_0)} \mid F_0] \leq \left(\exp\left(\frac{A_1}{c}(e^{tc} - 1) - tA_1\right) \;\middle|\; F_0\right). \tag{10.10}$$

Combining (10.9) and (10.10), we have

$$\mathbb{E}\left[e^{t(Z_n - Z_0)} \;\middle|\; F_0\right] \leq \left(\exp\left(\frac{\mu}{c}(e^{tc} - 1) - t\mu\right) \;\middle|\; F_0\right) = \exp\left(\frac{\mu}{c}(e^{tc} - 1) - t\mu\right).$$

□

Proof of Theorem 37. By Lemma 39 and the law of total expectation,

$$\begin{aligned}\mathbb{E}[e^{t(Z_n - Z_0)}] &= \mathbb{E}[\mathbb{E}[e^{t(Z_n - Z_0)} \mid F_0]] \\ &\leq \mathbb{E}\left[\exp\left(\frac{\mu}{c}(e^{tc} - 1) - t\mu\right)\right] \\ &= \exp\left(\frac{\mu}{c}(e^{tc} - 1) - t\mu\right). \end{aligned}$$

The rest of the proof is identical to the proof of Theorem 32. □

Corollary 38 is a straightforward application of Theorem 37.

Proof of Corollary 38. Define the filtration $F_0, F_1, \ldots, F_n$ by

$$F_i = \sigma(X_1, X_2, \ldots, X_i, \mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{i+1}).$$

That is, F_i is the smallest σ-algebra with respect to which all of $X_1, X_2, \ldots, X_i$ and $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{i+1}$ are measurable.

Define $A_i = \mathbb{E}[X_i \mid \mathcal{D}_i]$ to be the expected value of X_i once its distribution is determined, and $B_i = c - A_i$. Define $Z_0, \ldots, Z_n$ to be given by

$$Z_i = \sum_{j=1}^{i} X_i - \sum_{j=1}^{i} A_i.$$

Since A_i and B_i are D_i-measurable and F_{i-1} contains D_i, we know that A_i and B_i are also F_{i-1}-measurable, implying that they are predictable processes with respect to filtration $F_0, ..., F_n$.

As each X_i is drawn from distribution D_i after all of $X_1, ..., X_{i-1}$ and $D_1, ..., D_i$ have been determined, we have $\mathbb{E}[X_i \mid F_{i-1}] = \mathbb{E}[X_i \mid D_i]$. We can then compute that

$$\begin{aligned}
\mathbb{E}[Z_i \mid F_{i-1}] &= \mathbb{E}[X_i - A_i + Z_{i-1} \mid F_{i-1}] \\
&= \mathbb{E}[X_i \mid F_{i-1}] - A_i + Z_{i-1} \\
&= \mathbb{E}[X_i \mid D_i] - A_i + Z_{i-1} \\
&= Z_{i-1},
\end{aligned}$$

implying that $Z_0, ..., Z_n$ is a martingale with respect to filtration $F_0, ..., F_n$.

Finally, $\{Z_i\}$, $\{A_i\}$ and $\{B_i\}$ satisfy the requirements of Theorem 37, namely that $-A_i \leq Z_i - Z_{i-1} \leq B_i$, that $A_i + B_i = c$, and that $\sum_i A_i \leq \mu$. Thus, by Theorem 37,

$$\mathbb{P}[Z_n \geq \delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right).$$

Expanding out Z_n gives

$$\mathbb{P}\left[\sum_i X_i \geq \sum_i A_i + \delta\mu\right] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right),$$

and thus we have

$$\mathbb{P}\left[\sum_i X_i \geq (1+\delta)\mu\right] \leq \exp\left(-\frac{\delta^2\mu}{(2+\delta)c}\right),$$

as desired. □

Applications of Theorem 37 in Concurrent Work

By allowing for an adaptive adversary, Theorem 37 naturally lends itself to applications with online adversaries. We conclude the section by briefly discussing two applications of Theorem 37 that have arisen in several of our recent concurrent works. In both cases, Theorem 37 significantly simplified the task of analyzing an algorithm.

Edge orientation in incremental forests In [10], Bender et al. consider the problem of edge orientation in an incremental forest. In this problem, edges $e_1, e_2, \ldots, e_k$ of a forest arrive one by one, and we are responsible for maintaining an orientation of the edges (i.e., an assignment of directions to the edges) such that every vertex has out-degree at most $O(1)$. As each edge e_i arrives, we may need to flip the orientations of other edges in order to accommodate the newly arrived edge. The goal in [10] is to flip at most $O(\log\log n)$ orientations per edge insertion (with high probability). We refer to an edge insertion as a step.

A key component of the algorithm in [10] is that vertices may "volunteer" to have their out-degree incremented during a given step. During each step i, there are $\operatorname{polylog} n$ vertices S_i that are eligible to volunteer, and each of these vertices volunteers with probability $1/\operatorname{polylog} n$. The algorithm is designed to satisfy the property that each vertex v can appear in at most $O(\log n)$ S_i's.

An essential piece of the analysis is to show that, for any set S of size $\operatorname{polylog} n$,

the number of vertices in S that ever volunteer is at most $|S|/2$ (with high probability). On a given step i, the expected number of vertices in S that volunteer is $|S \cap S_i|/\operatorname{polylog} n$. S_i is partially a function of the algorithm's past randomness, and thus S_i are effectively selected by an adaptive adversary, subject to the constraint that each vertex v appears in at most $O(\log n)$ S_i's. By applying Theorem 37, one can deduce that the number of vertices in S that volunteer is small (with high probability).

Note that, since $|S| = \operatorname{polylog} n$, a bound with additive error would not suffice here. Such a bound would allow for the number of vertices that volunteer to deviate by $\Omega(\sqrt{n})$ from its mean, which is larger than $|S|/2$.

Task scheduling against an adaptive adversary Another concurrent work to ours [11] considers a scheduling problem in which the arrival of new work to be scheduled is controlled by a (mostly) adaptive adversary. In particular, although the amount of new work that arrives during each step is fixed (to $1 - \epsilon$), the tasks to which that new work is assigned are determined by the adversary. The scheduling algorithm is then allowed to select a single task to perform 1 unit of work on. The goal is to design a scheduling algorithm that prevents the backlog (i.e., the maximum amount of unfinished work for any task) from becoming too large.

Due to the complexity of the algorithm in [11], we cannot explain in detail the application of Theorem 37. The basic idea, however, is that the adversary must decide how to allocate its resources across tasks over time, but that the adversary can adapt (in an online fashion) to events that it has observed in the past. Theorem 37 allows for the authors of [11] to obtain Chernoff-style bounds on the number of a certain "bad events" that occur, while handling the adaptiveness of the adversary.

Chapter 11

Counterexample to the Naive Parallel Dijkstra's Algorithm

11.1 Introduction

The single-source shortest path (SSSP) problem on graphs with non-negative weights is an extremely well studied and practically important problem in graph theory. For a graph G with vertex set V, source vertex $s \in V$, edge set E, and edge weights $w : E \to \mathbb{R}^+$, the SSSP problem finds the length of the shortest path from s to every vertex $u \in V$, where the length of a path is the sum of the weights of its edges.

In the serial setting, Dijkstra's algorithm [32] using Fibonacci heaps [39] achieves the best known serial asymptotic running time of $O(|E| + |V| \log |V|)$ for general graphs. Better running times are known for special cases; for instance, [71] presents an algorithm with $O(|E| + |V|)$ running time for graphs with integer edge weights.

Finding an efficient parallel algorithm for the SSSP problem is notoriously difficult. The Δ-stepping algorithm presented in [61] has good empirical performance on many kinds of graphs, but has no good theoretical guarantees on general graphs. A number of other algorithms have been proposed, such as [16, 26, 29, 50, 68], but none are able to both match the optimal work bound of $O(|E| + |V| \log |V|)$ and achieve low span.

One intuitive solution to the SSSP problem directly parallelizes Dijkstra's algorithm. Instead of extracting the lowest cost vertex and processing it in each step, the

P lowest cost vertices can be extracted at the same time and processed in parallel on P processors. As we'll see in this chapter, this seemingly reasonable algorithm is in fact fundamentally flawed. I construct a class of graphs on which this parallel algorithm achieves no speedup over the serial Dijkstra's algorithm.

11.2 Naive Parallelization of Dijkstra's Algorithm

We start by recalling the serial version of Dijkstra's algorithm. The following pseudocode presents a slightly non-standard version of Dijkstra's algorithm that is convenient to parallelize.

Serial Dijkstra's Algorithm

```
 1: function DIJKSTRA(G, w, s)
 2:     for u ∈ G.V do
 3:         d[u] ← ∞
 4:     end for
 5:     d[s] ← 0
 6:     Q ← {s}
 7:     while Q ≠ ∅ do
 8:         u ← EXTRACT-MIN(Q)
 9:         for edges uv do
10:             RELAX(u, v, w, Q)
11:         end for
12:     end while
13: end function
14: function RELAX(u, v, w, Q)
15:     if d[v] > d[u] + w(uv) then
16:         d[v] ← d[u] + w(uv)
17:         Q.INSERT(v)                ▷ Reinsertion equivalent to no-op
18:     end if
19: end function
```

Dijkstra's algorithm keeps track of a measure of ***tentative distance***, d, and a priority queue Q of vertices to be visited. The algorithm proceeds in ***rounds***. In each round, the algorithm extracts the vertex u in Q with the lowest tentative distance, and relaxes all edges from u. If the length of the path to a neighbor v through u is shorter than the recorded tentative distance to v, then the tentative distance is

updated and v added to the priority queue of vertices to be examined. It is possible for a vertex already in the priority queue to be inserted again in this process; such a re-insert has no effect on the priority queue.

A vertex u is said to be ***settled*** if its tentative distance $d(u)$ equals the length of the true shortest distance path to u. Dijkstra's algorithm satisfies the key property that the vertex u with the lowest tentative distance extracted from Q in each round must be settled. This property is proven on [27, p. 660]. The tentative distance of a settled vertex cannot be further improved; as a result, a vertex that has been extracted in this algorithm is never inserted back into Q and extracted a second time. The algorithm is thus guaranteed to end after $|V|$ rounds.

Serial Dijkstra's algorithm can be parallelized by extracting up to P vertices in each round instead of a single vertex, where the extracted vertices are examined in parallel on P processors. The efficiency of the algorithm depends heavily on which vertices are extracted. Extracting and processing a vertex u that is not settled is a waste of work, since the tentative distance of u will later decrease, and vertex u will be added back into the queue and extracted again. The ideal algorithm only extracts settled vertices in each round.[1]

Unfortunately, it is difficult to determine which vertices, besides the one with the lowest tentative distance, are settled. The most intuitive and naive alternative is to extract the P vertices with the lowest tentative distances.

It turns out that this naive approach does not always work well. I will construct a family of graphs where this parallel approach performs poorly. When the algorithm is run on these graphs, the priority queue Q almost always contains a large number of settled vertices. Nonetheless, the algorithm is "tricked" into extracting the wrong vertices; only one vertex out of the P extracted vertices in each round is settled, and the rest of the work is wasted. As a result, the parallel Dijkstra's algorithm achieves almost no speedup over the serial Dijkstra's algorithm on these graphs.

[1]Note that some graphs do not benefit from this parallelization approach as there are not enough settled vertices in Q in each round. For instance, if the graph is simply a path with the source vertex s being one end of the path, then the priority queue Q never contains more than one vertex.

Naive Parallel Dijkstra's Algorithm

```
1: function DIJKSTRA(G, w, s)
2:     for u ∈ G.V do
3:         d[u] ← ∞
4:     end for
5:     d[s] ← 0
6:     Q ← {s}
7:     while Q ≠ ∅ do
8:         u_1, ··· , u_P ← EXTRACT-P-MIN(Q)        ▷ Extracts fewer if |Q| < P
9:         parallel for u_i ∈ {u_1, ··· , u_P} do
10:            for edges u_i v do
11:                RELAX(u, v, w, Q)
12:            end for
13:        end for
14:    end while
15: end function
16: function RELAX(u, v, w, Q)
17:    if d[v] > d[u] + w(uv) then
18:        d[v] ← d[u] + w(uv)
19:        Q.INSERT(v)                              ▷ Reinsertion equivalent to no-op
20:    end if
21: end function
```

11.3 Construction of Counterexample

Denote the family of graphs by $\mathcal{G} = \{G_1, G_2, \cdots\}$. G_k is constructed in two stages.

First, construct the subgraph A as shown in Figure 11-1. Subgraph A contains $2k+1$ vertices labelled $a_1, \cdots, a_{2k+1}$. Each vertex $a_i, 1 \leq i \leq 2k$ is connected to a_{i+1} by an edge of weight 1, and each vertex $a_{2i-1}, 1 \leq i \leq k$ is connected to a_{2i+1} by an edge of weight 3.

Next, using the subgraph A, construct G_k as shown in Figure 11-2. Start with a full and complete binary tree with k layers of edges. For edges in layer i, where edges incident to the root are in layer 1 and edges incident to the leaves are in layer k, give all left-pointing edges a weight of 0 and all right-pointing edges a weight of 2^{k-i}. Finally, attach subgraph A to every leaf node of this binary tree. The subgraph A attached to the ith leaf node (counting from the left, starting at 0) is labelled by A_i, and its vertices from top to bottom are labelled $a_1^i, a_2^i, \cdots, a_{2k+1}^i$.

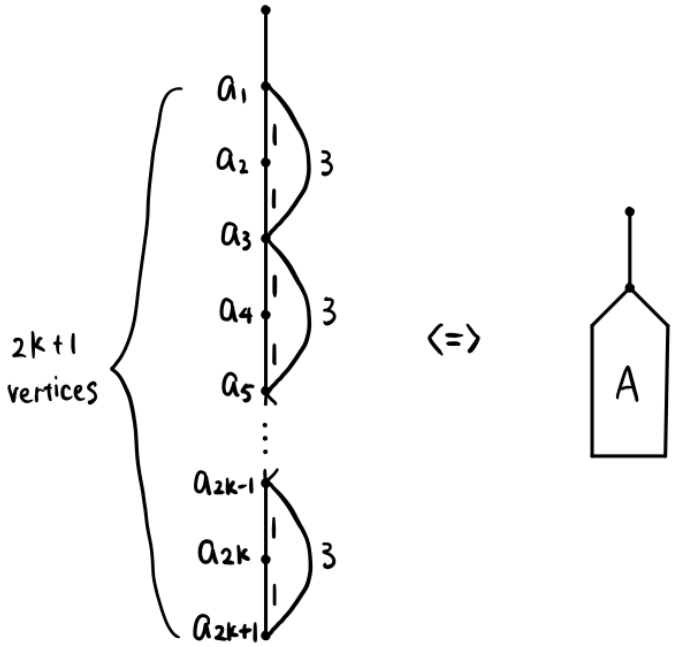

Figure 11-1: Subgraph A. The entire subgraph is represented by a tag with letter A inside.

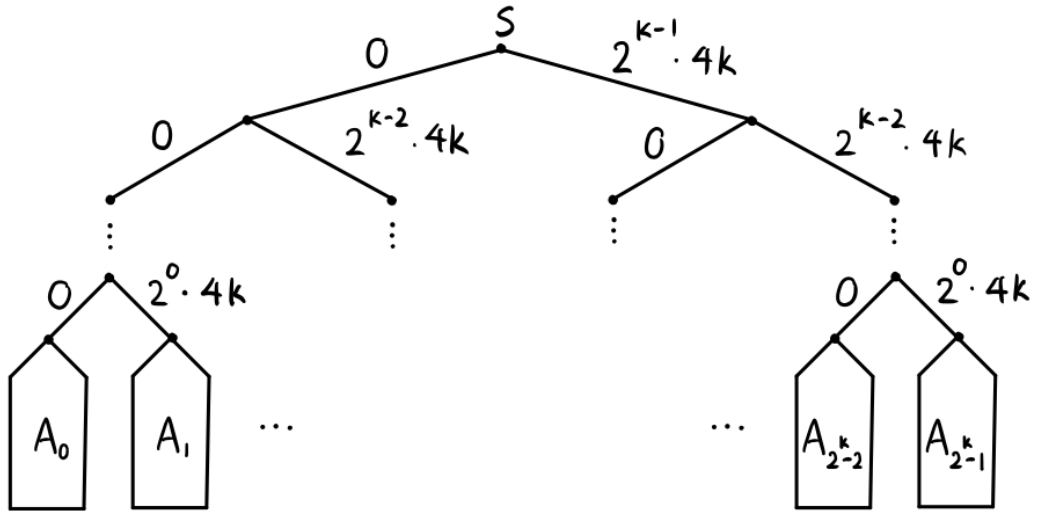

Figure 11-2: Graph G_k.

Note that this construction G_k has a number of desirable properties. It is sparse with a maximum degree of merely 4. It has a low diameter, and in fact does not contain any long, cycle-free paths at all. A longest path through this graph starts at the bottom of a subgraph A_i, passes through the root of the binary tree, then ends at the bottom of another subgraph A_j, for a total length of $\approx 6k = O(\log |V|)$.

Since G_k does not contain long paths, the parallel Dijkstra's algorithm is not bottlenecked by a lack of settled vertices in the priority queue Q. Nonetheless, we'll see that the algorithm almost never extracts more than one settled vertex in each round.

11.4 Proof that Naive Parallel Dijkstra's Algorithm Fails on Construction

This section proves that parallel Dijkstra's algorithm takes nearly $|V|$ rounds to complete. Since the serial Dijkstra's algorithm takes $|V|$ rounds to complete, this shows that the parallel algorithm achieves almost no speedup over its serial counterpart.

Theorem 40. For a fixed constant P, parallel Dijkstra's algorithm on P processors takes $|V| \cdot (1 - o(1))$ rounds to complete on the family of graphs G_k.

Intuition

Before diving into a precise proof, we'll see an intuitive explanation of why this theorem holds.

The A subgraphs are constructed such when the parallel Dijkstra's algorithm is run on just A, with the exception of P rounds to ramp up at the beginning and P rounds at the end, the priority queue Q always contains at least P vertices available to be extracted. However, the true shortest path to each vertex in A only contains edges of weight 1, and thus only one vertex in Q is actually settled in each step.

The binary tree in G_k ensures that the distances to the first vertex in different A subgraphs are far apart. Therefore, when the parallel Dijkstra's algorithm is run on

G_k, tentative distances to vertices in subgraph A_i are always all less than tentative distances to vertices in A_j for $i < j$.

As a result, even though multiple settled vertices belonging to different A subgraphs are available in Q, the algorithm generally only extracts vertices from one particular A subgraph and thus only extract a single settled vertex in each round. The algorithm ends after all $|V|$ vertices have been settled and extracted, and thus the parallel Dijkstra's algorithm takes close to $|V|$ rounds on G_k.

Proof

We start with Lemma 41, which observes important invariants about the algorithm's internal state. Recall that d denotes tentative distances.

Lemma 41. At the beginning of every round of the algorithm, for every $0 \leq i \leq 2^k - 1$, either $Q \cap A_i = \emptyset$, or the following invariants hold:

1. $Q \cap A_i$ consists of vertices $a_s^i, a_{s+1}^i, \cdots, a_t^i$ for some $1 \leq s \leq t \leq 2k+1$, t odd.
2. For $1 \leq r \leq s$, vertex a_r^i is settled.
3. For $s+1 \leq r \leq t$, we have $d[a_r^i] \geq d[a_{r-1}^i] + 2$.
4. For $t+1 \leq r \leq 2k+1$, we have $d[a_r^i] = \infty$.

Proof. Proceed by induction. For notational simplicity, A is written in place of A_i, and a_r in place of a_r^i.

Base case. The lemma holds up to and including the first round where $Q \cap A \neq \emptyset$.

Subproof. The lemma is trivially true when $Q \cap A = \emptyset$.

$Q \cap A$ first becomes non-empty when the incoming edge of a_1 in the binary tree is relaxed. When this happens, a_1 is added to Q, and since the path from s to a_1 is unique, $d[a_1]$ is updated to the correct shortest distance and a_1 is settled. All other $a_r, r \geq 2$ have not had an incident edge relaxed, and thus $d[a_r] = \infty$, $a_r \notin Q$. This satisfies all invariants. ■

Inductive step. Let the priority queue be Q and tentative distances be d at the beginning of round X. Let the priority queue be Q' and tentative distances be d' at the beginning of round $X+1$.

If $Q \cap A \neq \emptyset$ and the invariants hold for Q and d, then the invariants also hold for Q' and d'.

Subproof. Consider how $Q' \cap A$ and d' restricted on A change as a result of relaxing the incident edges of extracted vertices. Any vertex outside of A extracted in this round have no neighbors in A and thus have no effect on A.[2] Thus, we only need to consider what happens to vertices in A as a result of relaxing incident edges of extracted vertices in A.

Let $Q \cap A = \{a_s, ..., a_t\}$ by invariant 1. Suppose that in round X, the algorithm extracts and processes q vertices from $Q \cap A$. If $q = 0$, then nothing changed and the invariants continue to hold. Otherwise, assume $q \geq 1$. By invariant 3, the extracted vertices must be $a_s, a_{s+1}, \cdots, a_{s+q-1}$, as these vertices have the lowest tentative distances. Consider the following cases:

- **Case 1:** a_r for $1 \leq r \leq s$.

  By invariant 2, a_r was settled in round X and remains settled in round $X+1$. Moreover, if $1 \leq r \leq s-1$, then $a_r \notin Q$, and if $r = s$, then $a_r \in Q$ but is extracted in round X. In both cases, $a_r \notin Q'$ since a settled vertex cannot be re-inserted into the priority queue.

  Summary: a_r settled, $a_r \notin Q'$.

- **Case 2:** a_r for $r = s+1$.

  a_s was extracted and edge $a_s a_{s+1}$ with weight 1 relaxed. Since a_s was already settled and the shortest path to a_{s+1} does indeed contain edge $a_s a_{s+1}$, a_{s+1} becomes settled in round $X+1$. Since the tentative distance of a_{s+1} decreased, a_{s+1} is re-inserted into the priority queue.

[2] The only vertex outside of A with a neighbor in A is the vertex v in the binary tree adjacent to a_1. Since $Q \cap A \neq \emptyset$, a_1 must be settled and thus v must also be settled, so v cannot be extracted in this round.

Summary: a_r settled, $a_r \in Q'$.

- **Case 3:** a_r for $s+2 \leq r \leq \min(s+q, 2k+1)$, r even.

  The tentative distance of a_r was only updated through extracting a_{r-1} and relaxing the edge $a_{r-1}a_r$ with weight 1. Therefore, $d'[a_r] = d[a_{r-1}] + 1$. By invariant 3, $d[a_{r-1}]+1 \leq d[a_r]-1$. Therefore, $d'[a_r] \leq d[a_r]-1$, and a_r is (re)-inserted into the priority queue.

  Moreover, $d'[a_{r-1}] \leq d[a_{r-2}] + 1$ since the tentative distance of a_{r-1} is at least updated by relaxing $a_{r-2}a_{r-1}$. Therefore,

$$\begin{aligned} d'[a_r] &= d[a_{r-1}] + 1 \\ &\geq d[a_{r-2}] + 3 \quad \text{(invariant 3)} \\ &\geq d'[a_{r-1}] + 2 \end{aligned}$$

  Summary: $d'[a_r] \geq d'[a_{r-1}] + 2$, $a_r \in Q'$.

- **Case 4:** a_r for $s+2 \leq r \leq \min(s+q+1, 2k+1)$, r odd.

  The tentative distance of a_r was also updated in two ways: extracting a_{r-1} and relaxing the edge $a_{r-1}a_r$ with weight 1, and extracting a_{r-2} and relaxing the edge $a_{r-2}a_r$ with weight 3. By invariant 3, $d[a_{r-2}]+3 \leq d[a_{r-1}]+1$. Therefore, relaxing through edge $a_{r-2}a_r$ gives the stronger bound, and $d'[a_r] = d[a_{r-2}] + 3$. We also know that $d'[a_{r-1}] = d[a_{r-2}] + 1$, since the tentative distance of a_{r-1} was only updated through relaxing the edge $a_{r-2}a_{r-1}$. Therefore,

$$\begin{aligned} d'[a_r] &= d[a_{r-2}] + 3 \\ &= d'[a_{r-1}] + 2 \end{aligned}$$

  Summary: $d'[a_r] \geq d'[a_{r-1}] + 2$, $a_r \in Q'$.

- **Case 5:** a_r for $l \leq r \leq t$, where $l = \begin{cases} s+q+1 & \text{if } s+q+1 \text{ even} \\ s+q+2 & \text{if } s+q+1 \text{ odd} \end{cases}$ is the smallest

index not covered by case 3 or 4.

$a_r \in Q$ was not extracted and also did not have any incident edge relaxed. Therefore, a_r remains inside Q', and $d'[a_r] = d[a_r]$. We also know that $d'[a_{r-1}] \leq d[a_{r-1}]$ since tentative distances cannot increase. Therefore,

$$\begin{aligned} d'[a_r] &= d[a_r] \\ &\geq d[a_{r-1}] + 2 \quad \text{(invariant 3)} \\ &\geq d'[a_{r-1}] + 2 \end{aligned}$$

Summary: $d'[a_r] \geq d'[a_{r-1}] + 2$, $a_r \in Q'$.

- **Case 6:** a_r for $\max(t+1, l) \leq r \leq 2k+1$, where l is as defined in case 5.

  $a_r \notin Q$ did not have any incident edge relaxed. Therefore, a_r remains outside of Q', and $d'[a_r]$ remains at ∞.

  Summary: $d'[a_r] = \infty$, $a_r \notin Q'$.

Lastly, we combine the summaries of these cases.

As covered by cases 2 through 5, Q' contains $\{a_{s'}, \cdots, a_{t'}\}$ where $s' = s+1$ and $t' = \max(t, l-1, 2k+1)$. Since t is odd by invariant 1 and l is even by definition, we have that t' is odd. Case 1 and 2 state that for $1 \leq r \leq s'$, vertex a_r is settled. Cases 3, 4, and 5 state that for $s'+1 \leq r \leq t'$, we have that $d'[a_r] \geq d'[a_{r-1}] + 2$. Case 6 states that for $t'+1 \leq r \leq 2k+1$, we have that $d'[a_r] = \infty$. This concludes the proof of the inductive step.

■

Combining the base case and the inductive step concludes the proof of Lemma 41.

□

Corollary 42. Let the priority queue at the beginning of round X be Q, and let the priority queue at the beginning of round $X+1$ be Q'. Suppose that $q > 0$ vertices are extracted from $Q \cap A_i$ in round X, and a^i_{2k+1} is not among the vertices extracted. Then $|Q' \cap A_i| \geq \min(|Q \cap A_i| + 1, q)$.

Proof. We'll use Lemma 41 as well as the final part of proof of the inductive step. Let $Q \cap A_i = \{a_s, \cdots, a_t\}$ and $Q' \cap A_i = \{a_{s'}, \cdots, a_{t'}\}$. Note that $s' = s + 1$ and $t' \geq t$.

In the first case, $q < t-s+1$, or equivalently not all vertices in $Q \cap A_i$ are extracted in round X. In this case, $Q' \cap A_i$ contains at least $t'-s'+1 \geq t-(s+1)+1 = t-s \geq q$ vertices.

In the second case, $q = t-s+1$, or equivalently all vertices in $Q \cap A_i$ are extracted in round X. Since a_t is extracted and $a_t \neq a_{2k+1}$ is not the last vertex in A, it holds that $t' > t$. Since t' and t are both odd, $t' - t \geq 2$. In this case, $Q' \cap A_i$ contains at least $t' - s' + 1 \geq (t+2) - (s+1) + 1 = t - s + 2 = |Q \cap A_i| + 1$ vertices. □

Definition 43. At any point in the algorithm, denote by $count(A_i)$ the number of rounds that extracted a nonzero number of vertices from A_i.

Corollary 44. $Q \cap A_i = \emptyset$ for the current and all future rounds of the algorithm if and only if $count(A_i) = 2k + 1$.

Proof. We'll use Lemma 41 as well as the final part of proof of the inductive step. Consider the lowest-indexed vertex a^i_r contained in $Q \cap A_i$. When $Q \cap A_i$ first becomes non-empty, this lowest-indexed vertex is a^i_1. Each round that extracts a nonzero number of vertices from A_i increases this lowest index by 1. Once a^i_{2k+1} is extracted, all vertices in A_i are settled and will not be inserted into Q again. Therefore, $Q \cap A_i$ becomes and remains empty if and only if $2k + 1$ rounds have extracted a nonzero number of vertices from A_i. □

Definition 45. Subgraph A_i is said to be the ***active*** subgraph at the beginning of a round of the algorithm if i is the minimal index for which $count(A_i) < 2k + 1$.

In order to upper bound the total number of rounds of the algorithm, we'll upper bound the number of rounds for which A_i is the active graph for each fixed i. To do so, we need to investigate in what order extractions happen on different subgraphs A_i.

Lemma 46. In each round, the algorithm extracts those vertices from Q that belong to the leftmost subgraphs A_i. To be precise, the algorithm extracts up to P vertices from $Q \cap A_0$. If $|Q \cap A_0| < P$, it extracts up to the missing amount from $Q \cap A_1$. If still fewer than P vertices have been extracted, it extracts up to the missing amount from $Q \cap A_2$, and so on.

Proof. Observe that the length of the shortest path to a_1^i is $4k \cdot i$. In addition, every path within subgraph A has a length less than $4k$.

As a result, for $i < j$, it holds at the beginning of every round the tentative distances of all vertices in $Q \cap A_i$ are less than the tentative distances of all vertices in $Q \cap A_j$. Therefore, Extract-P-Min proceeds from left to right when extracting from subgraphs A_i. □

Lemma 47. The leaves of the binary tree are discovered from left to right. To be precise, if vertex a_1^i is added to Q in round X and vertex a_1^j is added to Q in round Y for $i < j$, then $X \leq Y$.

Proof. This holds by construction. For any two nodes in the tree on the same level, the shortest distance to the left node is less than the shortest distance to the right node. □

Lemma 48. For each A_i, among the rounds for which A_i is the active subgraph, there are at most $2P$ rounds where fewer than P vertices were extracted from A_i.

Proof. By Lemma 46, once A_i becomes the active subgraph, all future rounds extract as many vertices from A_i as possible. By Corollary 42, each of these rounds where A_i is the active subgraph increases $|Q \cap A_i|$, until it is guaranteed that $|Q \cap A_i| \geq P$ after P rounds. Every round starting from when $|Q \cap A_i| \geq P$ and ending at when a_{2k+1}^i is extracted extracts a full P vertices from A_i. a_{2k+1}^i can only be extracted if all but the last P vertices of A_i have settled. Therefore, at most P more rounds can take place before $count(A_i) = 2k + 1$ and A_i is no longer the active subgraph. □

Lemma 49. For each A_i, there are at most $2P^2$ rounds where A_i is not the active subgraph, but the algorithm extracts a nonzero number of vertices from A_i.

Proof. Suppose that A_m is the active subgraph at the beginning of some round. Lemmas 46 and 47 imply that the algorithm extracts vertices starting at A_m and move rightwards to other subgraphs as needed. Suppose that this round extracts a non-zero number of vertex from each of the subgraphs $A_m, A_{m+1}, \cdots, A_{m+l-1}$; since a round can extract at most P vertices total, $l \leq P$. As a result, the algorithm can extract a nonzero number of vertices from A_i only if the current active subgraph A_m satisfies that $m > i - P$.

In addition, the algorithm can only extract a nonzero number of vertices from A_i if it does not extract all P vertices from the active subgraph A_m. By Lemma 48, for each m, there are at most $2P$ rounds where not all P vertices are extracted from A_m.

Combining these two results implies that the algorithm can, at worst, spend $2P$ rounds extracting from A_i for each of the P values of m where A_m is the active subgraph. This results in a total of at most $2P^2$ rounds spent extracting from A_i while A_i is not the active subgraph. □

Finally, Corollary 44 and Lemma 49 combined proves Theorem 40. By Lemma 49, $count(A_i) \leq 2P^2$ when A_i first becomes the active subgraph. By Corollary 44, $count(A_i) = 2k + 1$ when A_i is no longer the active subgraph. Combining these two statements and using the fact that $count(A_i)$ can increment by at most 1 in each round, we find that the algorithm spends at least $2k + 1 - 2P^2$ rounds with A_i being the active subgraph for each A_i. As there are 2^k subgraphs A_i, the algorithm must take at least $2^k(2k + 1 - 2P^2)$ rounds.

Simple counting reveals that the graph contains $2^k(2k + 2) - 1$ vertices. We can thus write $2^k(2k + 1 - 2P^2) = 2^k(2k + 2)(1 - \frac{2P^2+1}{2k+2}) = |V|(1 - o(1))$, which proves Theorem 40.

Appendix A

The Stack-Augmented Split-Tree Mechanism without Synchronization Primitives

In this section, we describe how the dynamic array data structure and the stack-augmented split-tree mechanism can be modified to maintain correctness against determinacy races without using any form of synchronization primitives, such as locks, atomic compare-and-swaps, and more.

The dynamic array is modified to support a slightly different set of operations with different theoretical overheads, but with stronger guarantees when it comes to concurrent single-writer-multiple-reader scenarios.

The dynamic array stores entries of the form `(key, value)`, where the key is of a type that can be read and modified atomically. Moreover, there must exist some value `INV` that is never used as a valid key, and can be used to indicate that an entry is invalid. For leaf stack in the stack-augmented split-tree mechanism, the key has integer type, and -2 may be used as the invalid value.

The dynamic array supports the operations `create`, `append`, and `pop` in $O(1)$ time. It supports `destruct` in $O(\log n)$ time, where n is the length of the dynamic array being destroyed. It does not support `modify` at an arbitrary index, but does indirectly support `modify` at the *last* index using a `pop` followed by an `append`. It

also does not support `get` at an arbitrary index, at least not in constant time, but does support `get` at the *last* index in $O(1)$ time. It supports an additional operation `search` of an input key that replaces the need for `get` at arbitrary indices in the mechanism.

The modified dynamic array guarantees that the following key property holds:

Property 50. Consider a key k and a sequence of `append` and `pop` operations to a dynamic array performed sequentially by a single writer, such that at every point in the sequence of operations,

- the keys in the array are sorted and all distinct,
- the length of the array is greater than or equal to some value $l > 0$, and
- the key at index l is the largest key less than or equal to k.

Then, a reader performing a `search` for key k, concurrently with the writer performing this sequence of operations, will return the value at index l, no matter how the instructions from the reader and writer are interleaved.

The dynamic array makes use of linked lists. Each link in the linked list contains pointers to the previous and next link, and the list keeps track of the head and tail. This linked list is only modified by adding or removing a link at the tail end. One does not need to be careful about the exact order of modification of the next/previous links versus the tail pointer when it comes to updates.

The dynamic array stores a number of memory allocations in a linked list. Each link contains the pointer to the start of an allocation and the size of this allocation, and each allocation is double the size of the previous allocation in the list. Each allocation stores an array of `(key, value)` entries. Combining entries from allocations in the linked list order yields the entirety of the contents of the dynamic array.

In addition to the linked list, the dynamic array keeps track of two integer indices, `next-ind` and `inv-ind`. The `next-ind` keeps track of the next unused index in the second-to-last array (that is, the array in the link that is next to the tail in the list). The array in the last link is empty and the key of its first entry is guaranteed to be `INV`. `inv-ind` keeps track of the next index of the tail array that should be set to

invalid.

To help understand where one needs to be careful in the following operations, keep in mind that dynamic arrays are used in a single-writer multiple-reader situation, where the writer may call `append`, `pop`, and `get last`, and readers are only allowed to call `search`.

Operations are performed as follows:

Create

Allocate a small array (e.g., size 2) and set the key to `INV` for each of its entries. Allocate an array double the size of the first array, and set the key of its first entry to `INV`. Create a linked list from these two arrays. Set `next-ind` to 0 and `inv-ind` to 1.

Append

Write into the entry at index `next-ind` in the second-to-last array in the list. Increment `next-ind`.

If the entries of the last array have not all been set to invalid since `inv-ind` is not equal to the length of the last array, set the key in up to two more entries to `INV` and update `inv-ind` accordingly.

If the second-to-last array is now full, call `resize-up`.

Resize-up

Allocate a new array double the size of the array in the tail link. Set the key of its first entry to `INV`. Create a new link for this array and add it to the tail end of the linked list. Set `next-ind` to 0 and `inv-ind` to 1.

Pop

If the second-to-last array is empty, call `resize-down`.

Decrement `next-ind`, setting it to one less than the length of the second-to-last array if it was 0. Set the key of the entry at index `next-ind` in the second-to-last array to `INV`.

Resize-down

Remove the tail link of the linked list and free the array stored inside.[1] Set `inv-ind` to the length of the new last array.

Get last

Get the value stored in the entry at the index one before `next-ind` in the second-to-last array; if `next-ind` is 0, get the last element in the third-to-last array.

Search

This operation takes as input a key k and performs something akin to a binary search. It returns the value of the entry with the largest possible key less than or equal to k.

Start at the head of the linked list. If the key in the first entry of the array in the next link is valid and less than or equal to k, then move onto the next link of the linked list and repeat.

Otherwise, perform a binary search on the array in the current link to find the entry with the largest key less than or equal to k. The `INV` key is considered to be larger than all valid keys. Return the value of the entry found in the search.

Destruct

Free the allocation pointed to by each link in the list, then free the list itself.

Proof sketch of Property 50. Let the array piece that contains index l be in link m. Since the length of the array never drops below l throughout this sequence of operations, the entry in index l is never modified. Moreover, the length of the linked list never drops below $m+1$, link m's pointer to its next link is never modified, and link $m+1$'s pointer to its allocated array is never modified. Throughout these operations, the first key of the array in link $m+1$ is either larger than k or `INV`. Therefore,

[1] Under the current scheme, it's possible for a sequence of alternating `append` and `pop` operations to cause repeated allocations and deallocations. This can be easily fixed, but as it has no impact on theoretical performance, we do not worry about this optimization.

`search` will correctly end at link m. The binary search within the array in link m will correctly find the entry at index l, as the key of every entry past index l will be either larger than k or `INV` throughout these operations. □

The splitter protocol is effectively unmodified. A primary access of a splitter will directly call `search` instead of manually performing a binary search.

Property 50 replaces Section 7.4 in showing that concurrent operations are safe. Analysis of the span of the execution trace needs to take into account an additional $O(\log n)$ term at return-from-spawn nodes, as `destruct` on a dynamic array of up to length n takes span up to $O(\log n)$ instead of $\Theta(1)$. This does not impact the analysis, and the results in Section 8 still hold.

Appendix B

Lower Tail Bounds for the Multiplicative Azuma's Inequality

In this section we prove a lower tail bound with multiplicative error for both the normal and the adversarial setting. Whereas Theorem 32 and Theorem 37 allow us to bound the probability of a random variable substantially exceeding its mean, Theorem 51 and Theorem 53 allow us to bound the probability of a random variable taking a substantially smaller value than its mean.

Theorem 51. Let $Z_0, Z_1, \ldots, Z_n$ be a submartingale, meaning that $\mathbb{E}[Z_i \mid Z_0, \ldots, Z_{i-1}] \geq Z_{i-1}$. Assume additionally that $-a_i \leq Z_i - Z_{i-1} \leq b_i$, where $a_i + b_i = c$ for some constant $c > 0$ independent of i. Let $\mu = \sum_{i=1}^{n} a_i$. Then for any $0 \leq \delta < 1$,

$$\mathbb{P}[Z_n - Z_0 \leq -\delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{2c}\right).$$

Corollary 52. Let $X_1, \ldots, X_n \in [0, c]$ be real-valued random variables with $c > 0$. Suppose $\mathbb{E}[X_i \mid X_1, \ldots, X_{i-1}] \geq a_i$ for all i. Let $\mu = \sum_{i=1}^{n} a_i$. Then for any $0 \leq \delta < 1$,

$$\mathbb{P}\left[\sum_i X_i \leq (1-\delta)\mu\right] \leq \exp\left(-\frac{\delta^2\mu}{2c}\right).$$

Theorem 53. Let $Z_0, \ldots Z_n$ be a submartingale with respect to the filtration $F_0, \ldots, F_n$, and let $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ be predictable processes with respect to the same filtration. Suppose there exist values $c > 0$ and μ, satisfying that $-A_i \leq Z_i - Z_{i-1} \leq B_i$, $A_i + B_i = c$, and $\sum_{i=1}^{n} A_i \geq \mu$ (almost surely). Then for any $\delta > 0$,

$$\mathbb{P}[Z_n - Z_0 \leq -\delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{2c}\right).$$

Corollary 54. Suppose that Alice constructs a sequence of random variables $X_1, \ldots X_n$, with $X_i \in [0, c], c > 0$, using the following iterative process. Once the outcomes of $X_1, \ldots, X_{i-1}$ are determined, Alice then selects the probability distribution $\mathcal{D}_i$ from which X_i will be drawn; X_i is then drawn from distribution $\mathcal{D}_i$. Alice is an adaptive adversary in that she can adapt $\mathcal{D}_i$ to the outcomes of $X_1, \ldots, X_{i-1}$. The only constraint on Alice is that $\sum_i \mathbb{E}[X_i \mid \mathcal{D}_i] \geq \mu$, that is, the sum of the means of the probability distributions $\mathcal{D}_1, \ldots, \mathcal{D}_n$ must be at least μ.

If $X = \sum_i X_i$, then for any $\delta > 0$,

$$\mathbb{P}[X \leq (1 - \delta)\mu] \leq \exp\left(-\frac{\delta^2\mu}{2c}\right).$$

We begin by proving Theorem 51. The proof is similar to the proof for the upper tail bound, with a different approximation used.

Lemma 55. For any $t < 0$ and any random variable X such that $\mathbb{E}[X] \geq 0$ and $-a \leq X \leq b$,

$$\mathbb{E}[e^{tX}] \leq \exp\left(\frac{a}{a+b}\left(e^{t(a+b)} - 1\right) - ta\right).$$

Proof. Same as Lemma 34. □

Proof of Theorem 51. By Markov's inequality, for any $t < 0$ and v,

$$\begin{aligned}\mathbb{P}[Z_n - Z_0 \leq v] &= \mathbb{P}[t(Z_n - Z_0) \geq tv] \\ &= \mathbb{P}\left[e^{t(Z_n - Z_0)} \geq e^{tv}\right] \\ &\leq \frac{\mathbb{E}[e^{t(Z_n - Z_0)}]}{e^{tv}}.\end{aligned}$$

Let $X_i = Z_i - Z_{i-1}$. Since Z_i is a submartingale, for any i, $\mathbb{E}[X_i \mid Z_0, \ldots, Z_{i-1}] \geq 0$. Moreover, from the assumptions in the problem, $-a_i \leq X_i \leq b_i$. Therefore, Lemma 55 applies to $X = (X_i \mid Z_0, \ldots, Z_{i-1})$, and we have

$$\mathbb{E}[e^{tX_i} \mid Z_0, \ldots, Z_{i-1}] \leq \exp\left(\frac{a_i}{c}\left(e^{tc} - 1\right) - ta_i\right),$$

for any $t < 0$. Using the same derivation as in the proof for Theorem 32, we have

$$\mathbb{P}[Z_n - Z_0 \leq v] \leq \exp\left(\frac{\mu}{c}\left(e^{tc} - 1\right) - t\mu - tv\right).$$

Plugging in $t = \ln(1-\delta)/c$ and $v = -\delta\mu$ for $\delta > 0$ yields

$$\mathbb{P}[Z_n - Z_0 \leq -\delta\mu] \leq \exp\left(\frac{\mu}{c}(-\delta - (1-\delta)\ln(1-\delta))\right).$$

For any $0 \leq \delta < 1$,

$$-\delta - (1-\delta)\ln(1-\delta) \leq -\frac{\delta^2}{2},$$

which can be seen by inspecting the derivative of both sides.[1] As a result,

$$\mathbb{P}[Z_n - Z_0 \leq -\delta\mu] \leq \exp\left(-\frac{\delta^2\mu}{2c}\right).$$

□

Remark. As with the upper tail bound, we may derive a stronger but more unwieldy

[1] Consider $f(x) = -x/(1-x) - \ln(1-x) + x^2/(2(1-x))$. Then $f(0) = 0$, and $f'(x) = -x^2/(2(1-x)^2) \leq 0$ for $0 \leq x < 1$. Therefore, $f(x) \leq 0$ for $0 \leq x < 1$, and the inequality holds for $0 \leq \delta < 1$.

bound of

$$\mathbb{P}[Z_n - Z_0 \leq -\delta\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu/c}.$$

The proof of Corollary 52 is identical to the proof of Corollary 33.

The proof of Theorem 53 can be obtained by combining the proofs of Theorem 37 and Theorem 51.

The proof of Corollary 54 is identical to the proof of Corollary 38.

Bibliography

[1] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.

[2] James Aspnes. Randomized consensus in expected $O(n^2)$ total work using single-writer registers. In *International Symposium on Distributed Computing*, pages 363–373. Springer, 2011.

[3] Vincenzo Auletta, Ioannis Caragiannis, Christos Kaklamanis, and Pino Persiano. Randomized path coloring on binary trees. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 60–71. Springer, 2000.

[4] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[5] Yossi Azar, Ilan Reuven Cohen, and Iftah Gamzu. The loss of serving in the dark. In *Forty-Fifth Annual ACM Symposium on Theory of Computing*, pages 951–960, 2013.

[6] Kazuoki Azuma. Weighted sums of certain dependent random variables. *Tohoku Mathematical Journal, Second Series*, 19(3):357–367, 1967.

[7] Michael A Bender, Jake Christensen, Alex Conway, Martin Farach-Colton, Rob Johnson, and Meng-Tsung Tsai. Optimal ball recycling. In *Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2527–2546. SIAM, 2019.

[8] Michael A Bender, Martín Farach-Colton, and William Kuszmaul. Achieving optimal backlog in multi-processor cup games. In *Fifty-First Annual ACM SIGACT Symposium on Theory of Computing*, pages 1148–1157, 2019.

[9] Michael A Bender, Tsvi Kopelowitz, William Kuszmaul, and Seth Pettie. Contention resolution without collision detection. In *Fifty-Second Annual ACM SIGACT Symposium on Theory of Computing*, pages 105–118, 2020.

[10] Michael A Bender, Tsvi Kopelowitz, William Kuszmaul, Eli Porat, and Clifford Stein. Edge orientation for incremental forests and low-latency cuckoo hashing. Under Submission to Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms, 2020.

[11] Michael A Bender and William Kuszmaul. Randomized cup game algorithms against strong adversaries. Under Submission to Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms, 2020.

[12] George Bennett. Probability inequalities for the sum of independent random variables. *Journal of the American Statistical Association*, 57(297):33–45, 1962.

[13] Sergei Bernstein. On a modification of Chebyshev's inequality and of the error formula of Laplace. *Ann. Sci. Inst. Sav. Ukraine, Sect. Math*, 1(4):38–49, 1924.

[14] Sergei N Bernstein. On certain modifications of Chebyshev's inequality. *Doklady Akademii Nauk SSSR*, 17(6):275–277, 1937.

[15] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, 2008.

[16] Guy E Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest paths using radius stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 443–454, 2016.

[17] 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.

[18] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[19] Béla Bollobás, Christian Borgs, Jennifer T Chayes, and Oliver Riordan. Directed scale-free graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, volume 3, pages 132–139, 2003.

[20] Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press, 2013.

[21] Arnaud Casteigts, Yves Métivier, John Michael Robson, and Akka Zemmari. Design patterns in beeping algorithms: Examples, emulation, and analysis. *Information and Computation*, 264:32–51, 2019.

[22] Nicolo Cesa-Bianchi, Alex Conconi, and Claudio Gentile. On the generalization ability of on-line learning algorithms. *IEEE Transactions on Information Theory*, 50(9):2050–2057, 2004.

[23] Deeparnab Chakrabarty and C Seshadhri. A $\tilde{O}(n)$ non-adaptive tester for unateness. *arXiv preprint arXiv:1608.06980*, 2016.

[24] Herman Chernoff. A career in statistics. *Past, Present, and Future of Statistical Science*, 29, 2014.

[25] Fan Chung and Linyuan Lu. Concentration inequalities and martingale inequalities: a survey. *Internet Mathematics*, 3(1):79–127, 2006.

[26] Edith Cohen. Using selective path-doubling for parallel shortest-path computations. *Journal of Algorithms*, 22(1):30–56, 1997.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[28] Kevin P Costello, Asaf Shapira, and Prasad Tetali. Randomized greedy: new variants of some classic approximation algorithms. In *Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 647–655. SIAM, 2011.

[29] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, pages 722–731. Springer, 1998.

[30] Victor H de la Pena, Michael J Klass, and Tze Leung Lai. Self-normalized processes: exponential inequalities, moment bounds and iterated logarithm laws. *Annals of Probability*, pages 1902–1933, 2004.

[31] Paul F Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, pages 67–74. Springer, 1989.

[32] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[33] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 202–210, 2003.

[34] Devdatt P Dubhashi and Desh Ranjan. Balls and bins: A study in negative dependence. *BRICS Report Series*, 3(25), 1996.

[35] John C Duchi, Alekh Agarwal, and Martin J Wainwright. Dual averaging for distributed optimization: Convergence analysis and network scaling. *IEEE Transactions on Automatic Control*, 57(3):592–606, 2011.

[36] Kacha Dzhaparidze and JH Van Zanten. On Bernstein-type inequalities for martingales. *Stochastic Processes and Their Applications*, 93(1):109–117, 2001.

[37] Xiequan Fan, Ion Grama, Quansheng Liu, et al. Exponential inequalities for martingales with applications. *Electronic Journal of Probability*, 20, 2015.

[38] Mingdong Feng and Charles E Leiserson. Efficient detection of determinacy races in cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[39] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[40] David A Freedman. On tail probabilities for martingales. *Annals of Probability*, pages 100–118, 1975.

[41] 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, page 79–90, New York, NY, USA, 2009. Association for Computing Machinery.

[42] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[43] Ajay Gopinathan and Zongpeng Li. Strategyproof mechanisms for content delivery via layered multicast. In *International Conference on Research in Networking*, pages 82–96. Springer, 2011.

[44] L Gyorfi, Gábor Lugosi, and Gusztáv Morvai. A simple randomized algorithm for sequential prediction of ergodic time series. *IEEE Transactions on Information Theory*, 45(7):2642–2650, 1999.

[45] Erich Haeusler. An exact rate of convergence in the functional central limit theorem for special martingale difference arrays. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 65(4):523–534, 1984.

[46] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426. Springer, 1994.

[47] Kumar Joag-Dev and Frank Proschan. Negative association of random variables with applications. *The Annals of Statistics*, pages 286–295, 1983.

[48] Rasul A Khan. Lp-version of the dubins–savage inequality and some exponential inequalities. *Journal of Theoretical Probability*, 22(2):348, 2009.

[49] Alam Khursheed and KM Lai Saxena. Positive dependence in multivariate distributions. *Communications in Statistics-Theory and Methods*, 10(12):1183–1196, 1981.

[50] Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

[51] Dennis Komm, Rastislav Kràlovic, Richard Kràlovic, and Tobias Mömke. Randomized online algorithms with high probability guarantees. In *Thirty-First International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[52] V Kumar. An approximation algorithm for circular arc colouring. *Algorithmica*, 30(3):406–417, 2001.

[53] I-Ting Angelina Lee and Tao B Schardl. Efficiently detecting races in cilk programs that use reducer hyperobjects. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 111–122, 2015.

[54] 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*, pages 287–297, 2012.

[55] Charles E Leiserson and Tao B Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 303–314, 2010.

[56] Emmanuel Lesigne and Dalibor Volnỳ. Large deviations for martingales. *Stochastic Processes and Their Applications*, 96(1):143–159, 2001.

[57] Reut Levi, Dana Ron, and Ronitt Rubinfeld. Local algorithms for sparse spanning graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2014)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[58] Robert Liptser and Vladimir Spokoiny. Deviation probability bound for martingales with applications to statistical estimation. *Statistics & probability letters*, 46(4):347–357, 2000.

[59] Quansheng Liu and Frédérique Watbled. Exponential inequalities for martingales and asymptotic properties of the free energy of directed polymers in a random environment. *Stochastic processes and their applications*, 119(10):3101–3132, 2009.

[60] Colin McDiarmid. On the method of bounded differences. *Surveys in Combinatorics*, 141(1):148–188, 1989.

[61] Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[62] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, November 2018. Version 5.0.

[63] Iosif Pinelis. Optimum bounds for the distributions of martingales in Banach spaces. *The Annals of Probability*, pages 1679–1706, 1994.

[64] Emmanuel Rio et al. Extensions of the Hoeffding-Azuma inequalities. *Electronic Communications in Probability*, 18, 2013.

[65] Emmanuel Rio et al. On McDiarmid's concentration inequality. *Electronic Communications in Probability*, 18, 2013.

[66] Sebastien Roch. Modern Discrete Probability: An Essential Toolkit. *Lecture notes*, 2015.

[67] Tao B Schardl, I-Ting Angelina Lee, and Charles E Leiserson. Brief announcement: Open cilk. In *30th on Symposium on Parallelism in Algorithms and Architectures*, pages 351–353, 2018.

[68] Hanmao Shi and Thomas H Spencer. Time–work tradeoffs of the single-source shortest paths problem. *Journal of algorithms*, 30(1):19–32, 1999.

[69] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts essentials*. Wiley Hoboken, 2018.

[70] Milan Straka. Optimal worst-case fully persistent arrays. *Trends in Functional Programming*, 2009.

[71] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[72] Sara A van de Geer. On Hoeffding's inequality for dependent random variables. In *Empirical Process Techniques for Dependent Data*, pages 161–169. Springer, 2002.

[73] David Wajc. Negative association: definition, properties, and applications. *Manuscript, available from https://goo. gl/j2ekqM*, 2017.

[74] Grigory Yaroslavtsev and Samson Zhou. Approximate F_2-sketching of valuation functions. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.