Summer 8-15-2017

Easier Parallel Programming with Provably-Efficient Runtime Schedulers

Robert Utterback
Washington University in St. Louis

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Kunal Agrawal, Co-Chair,
I-Ting Angelina Lee, Co-Chair
Sanmay Das
Jeremy T. Fineman
Christopher Gill

Easier Parallel Programming with Provably-Efficient Runtime Schedulers

by

Robert Utterback

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2017
Saint Louis, Missouri

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation would not have been possible without the help of numerous people. This section is not enough to convey my gratitude to these people, but it must suffice for now. I apologize in advance to anyone who feels they should have been mentioned; you are almost surely correct.

First, thanks to my advisers, Kunal Agrawal and Angelina Lee. Either one alone would be a great adviser; having both as mentors, teachers, and collaborators all but ensured success in finishing my Ph.D. They provided a balance of guidance and independence that shaped my development as a researcher.

It was Kunal who initially convinced me to attend WUSTL. Because I came in with no previous research experience, much of what she said flew right over my head, but I learned a lot trying to keep up. I may never understand how she can come up with the perfect structure for a presentation in just a few seconds, though.

A few years later Angelina became my co-adviser, providing a complimentary advising style. Our various discussions have ingrained in me nuggets of wisdom ranging from performance

engineering to teaching to general life advice. During the first few months my systems programming chops seemingly grew exponentially as we performance engineered various systems.

I would like to thank the other members of my committee: Sanmay Das, Chris Gill, and Jeremy Fineman. Jeremy also served as an excellent collaborator on many of the projects presented here. Speaking of collaborators, I would also like to thank Milind Kulkarni for his work on the processor-oblivious record and replay paper and Kefu Lu, Brendan Sheridan, and Jim Sukha for their contributions to the Batcher paper. Thanks to everyone I met during my internship at Huawei, who provided me with a pleasant taste of industry.

Thanks to the members of the parallel computing group at WUSTL. Thanks to Jordyn Maglalang for all the fruitful discussions that prevented me from wasting hours debugging my code. Thanks to David Ferry for showing me the ropes with regard to our servers. Thanks to Jing Li for many great discussions on Cilk, work stealing, and other research topics. Thanks to Ramsay Shuck and Yifan Xu for many discussions and allowing me to practice my mentoring and teaching skills on them. I hope the parallel computing group continues to grow in the coming years.

Outside of the parallel computing group I was fortunate to have great graduate student friends to spend time with. Many of the graduate students in the program came to my practice talks and empathized with me about the trials of graduate school. These include Sam Powell, Jon Shidal, Michelle Ichinco, Steve Cole, Ian Schillebeeckx, Missael García, Meenal Burrows, and countless others. A special shout-out to Adam Drescher, who became

a groomsmen in my wedding. Whether it was discussing half-baked research ideas or strange politics, doing so over a beer at Three Kings usually cured my ails. Thanks also to my friends outside the program, especially my Phi Kappa Tau brothers, for when I needed a break from academia.

I would like to express my gratitude toward the administrative staff who helped me throughout my time in the program: Monét Demming, Kelli Eckman, Myrna Harbison, Sharon Matlock, Cheryl Newman, Lauren Huffman, and Madeline Hawkins. Even when busy with other things, they were invariably ready to help when I needed it.

As if graduate school did not provide enough to do, I also got married during my time here. One would think this would extend my time in school, but Audrey actually made the whole experience easier. When I think I am too tired, she knows how to motivate me to work harder. When I actually am too tired, she knows when to take a break with me. During the most stressful times she takes up the entirety of the household duties without complaints. At all times she is supportive and caring.

Finally, I thank my parents, Steve and Debby, and my sisters, Amanda and Lora. I am not sure I ever properly explained to them what I have been "researching" in graduate school, but at least my attempts to explain it have improved over the years. Without their questions and gentle prodding I'm not sure I ever would have finished. In fact, I never would have made

it to higher education at all without my parents' efforts to instill in me a love of learning and provide me with a supportive environment.

Robert Utterback

Washington University in Saint Louis

August 2017

ABSTRACT OF THE DISSERTATION

Easier Parallel Programming with Provably-Efficient Runtime Schedulers

by

Robert Utterback

Doctor of Philosophy in Computer Science

Washington University in St. Louis, August 2017

Associate Professor Kunal Agrawal, Co-Chair

Assistant Professor I-Ting Angelina Lee, Co-Chair

Over the past decade processor manufacturers have pivoted from increasing uniprocessor performance to multicore architectures. However, utilizing this computational power has proved challenging for software developers. Many concurrency platforms and languages have emerged to address parallel programming challenges, yet writing correct and performant parallel code retains a reputation of being one of the hardest tasks a programmer can undertake.

This dissertation will study how runtime scheduling systems can be used to make parallel programming easier. We address the difficulty in writing parallel data structures, automatically finding shared memory bugs, and reproducing non-deterministic synchronization bugs. Each of the systems presented depends on a novel runtime system which provides strong theoretical performance guarantees and performs well in practice.

Chapter 1

Introduction

Until the beginning of the 21st century programmers tasked with improving performance enjoyed a special advantage. During that time processor performance improved rapidly, with chip speed doubling approximately every 18 months. Rather than spend time designing new algorithms or performance engineering, programmers could simply wait until the next chip came out. By simply replacing their current hardware with the new chip, performance would automatically improve.

By 2005 it was clear this "free lunch" was over [175]. Faced with problems with heat, power consumption, and current leakage, processor manufacturers stalled in their efforts to increase clock speeds and uniprocessor instruction throughput. Instead, they have increasingly turned to multicore architectures, as shown in figure 1.1. Compare, for example, two of Intel's desktop processors: the Pentium 4 660, released in 2005, and the Core i9-7900X, released in 2017 (data from the Intel ARK Database [52]). The 660 runs a single core at 3.6 GHz, while the i9 runs 10 cores at 3.3 GHz. The benefits obtained from these new parallel architectures depend on writing effective parallel software.

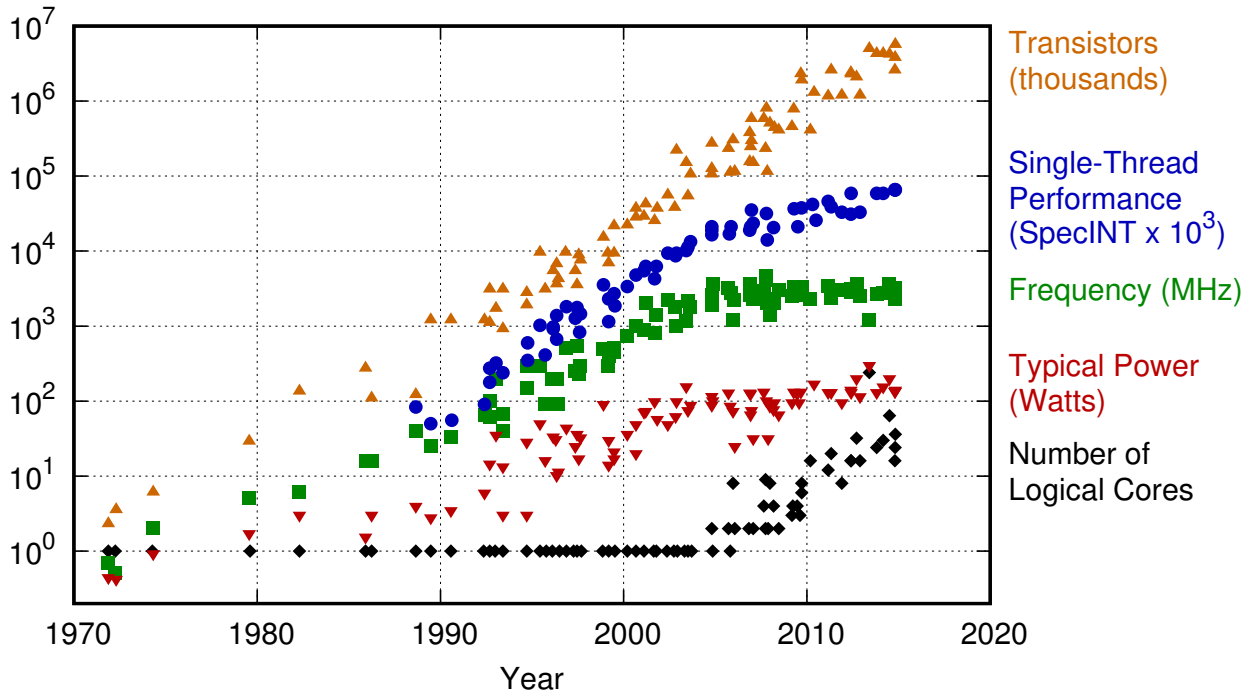


Figure 1.1: Multiprocessor trends over over the past 40 years. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New plot and data collected by K. Rupp [159]

Unfortunately, writing correct and efficient parallel programs is challenging. For years parallel software was written by programming with **persistent threads**, such as POSIX threads [98] ("pthreads"), Windows threads [87], and Java threads [85]. Programming to these threading APIs requires manually managing low-level scheduling, synchronization and task decomposition. Such details often require brittle boiler-plate code and tend to become intertwined with program logic. Combined with the nasty non-determinism that often comes from using shared memory in parallel and it is no wonder that parallel programming has a reputation of low productivity and hopelessness [130, 94, 95].

The need to more easily use multicore hardware has driven the development of **concurrency platforms**. The goal of such systems — which may be language libraries, language extensions, or even entire programming languages — is to provide an simpler abstraction for writing efficient parallel software. Popular systems include Cilk Plus [99], Intel TBB [181], OpenMP 3.0 [143], Habanero Java [40], Task Parallel Library [122], Chapel [41], and X10 [43], among others.

These concurrency platforms provide a programming abstraction known as **dynamic multithreading** (sometimes called "dthreading"), wherein the programmer uses parallel primitives to specify pieces of the program that are logically parallel. The platform relies on a runtime system to map this logical parallelism to the hardware at runtime. Such systems are **processor-oblivious**, meaning that they do not depend on any particular hardware configuration. Programmers using such systems need not be concerned with scheduling, load balancing, or the underlying hardware.

At the same time, we have seen the development of a set of dynamic tools to improve the experience of parallel programming. These include things like bug detectors (e.g. [166, 103, 152, 72, 164]), record and replay systems (for replaying specific non-deterministic executions, e.g. [146, 6, 125, 63]), profilers and performance tuning tools (e.g. [165, 176, 177, 102]. These tools operate at runtime, performing extra work while the program executes to accomplish the goals of the tool. These tools are useful even within the realm of concurrency platforms, as dynamic multithreading platforms do not entirely erase the burden of shared memory or performance tuning.

1.1 Contributions

This dissertation investigates how runtime schedulers can be co-designed with programming tools to improve the experience of parallel programming. The focus is on developing tools to help handle shared memory, which is often a source of insidious non-determinism. The process of scheduler/tool co-design also produces new insights for new schedulers that can extend the classes of computations normally considered by concurrency platforms.

This dissertation discusses the following projects:

- Batcher [4], which leverages a runtime scheduler to efficiently and transparently coordinate concurrent operations to a shared data structure.
- CRacer [185], an algorithm for detecting shared memory bugs that, with the help of the runtime scheduler, is asymptotically optimal and runs in parallel
- PORRidge [184], a system that eases debugging by recording some non-deterministic events and later replaying them in their original order.
- Two algorithms for detecting shared memory bugs in a larger class of computations than previous work has considered. The desire to implement these algorithms in the future led us to extend the Cilk Plus [99] platform to handle this larger class of computations.

The common element among the tools is that each leverages support from novel runtime scheduler. These runtime schedulers allow our tools to provide provable performance guarantees. Additionally, we implemented these systems and showed that each performs well in practice.

1.2 Outline

The rest of this dissertation is structured as follows. We first discuss some necessary background material used through this dissertation in chapter 2. Chapter 3 describes a runtime scheduler that provides a form of automatic synchronization in parallel programs that use data structures, while chapter 4 uses a variant of that runtime to provide asymptotically optimal detection of shared memory bugs. Chapter 5 presents a work-stealing runtime system that provides non-blocking support for handling asynchronous events in parallel program. We leverage that runtime to develop a tool to help debug non-deterministic bugs in chapter 6. We present recent work on detecting shared memory bugs in a very large class of computations in chapters 7 and 8. Chapter 9 concludes with some final thoughts on these tools and future work in this direction.

Chapter 2

Key Concepts in Dynamic Multithreading

This chapter sets the context for the projects in this dissertation by presenting the dynamic multithreading model. Traditionally, parallel software has been written using **static threading**, in which programmers manage virtual processors — **threads** — and manually partition work among them. In contrast, in **dynamic multithreading** the programmer specifies the logical parallelism of the program using primitives such as spawn/sync, async/finish, or parallel-for loops. At runtime, a scheduler is responsible for efficiently mapping this parallelism to the processing cores[1]. This class of programming languages includes the Cilk family [31, 77, 100], subsets of OpenMP [12], Threading Building Blocks [181], the Habanero family [15, 40], Task Parallel Library [121], X10 [40, 43], and many others.

We begin by describing how we model (section 2.1) and analyze (section 2.2) dynamically multithreaded programs. We discuss how dynamically multithreaded computations can be scheduled using work stealing in section 2.3. Section 2.4 presents the linguistic primitives

[1]We will generally use the terms **core** and **processor** interchangeably in this dissertation.

supported by Cilk, whose model forms the basis of much of this dissertation. Finally, we provide a brief discussion on the issue of determinacy in section 2.5.

2.1 DAG Model

The execution of a dynamically multithreaded program can be modeled as a directed acyclic graph (**DAG**) (see [51, Ch. 27]).Nodes in such a DAG represent **strands**, a chain of sequential instructions without any parallel constructs. We can split chains of instructions into strands in any way that is convenient for us. Often for theoretical analysis, for example, strands will represent single instructions, whereas a Cilk programmer might think of strands as *maximal* chains of instructions without Cilk primitives. However, strands will always respect function boundaries — each strand belongs to a single function instance. A parallel control dependency between strands u and v is presented as an edge $(u, v$ in the DAG. In other words, v cannot execute until after u has completed.

The DAG represents a **computation**, which we formally define as the *executed* (not source code) instructions of a program given a specific input. Importantly, the DAG unfolds dynamically — strands and dependencies are not known ahead of time.

We call nodes with out-degree two **spawn** (sometimes **fork**) strands, while nodes with at least two incoming edges are **sync** (sometimes **join**) strands. Without loss of generality, this dissertation makes a few assumptions about the structure of strands. We assume that a sync strand does not immediately follow a spawn strand, and we assume binary forking — strands can have out-degree no greater than two. As Blumofe [28] notes, DAGs can be converted to binary forking.

As mentioned above, for our theoretical analyses we will define strands as unit-time computations — a single instruction on an ideal parallel computer where each instruction takes unit time to execute. Along with this assumption we will also ignore memory bandwidth and cache effects.

2.2 Analysis of Dynamically Multithreaded Computations

There are two key metrics of a DAG A: its **work**, denoted $T_1(A)$, is the time to execute all strands in A. Since we typically assume that strands represent unit-time instructions, this is just the total number of nodes in the DAG. The **span** (also **critical-path length** or **depth** in the literature), denoted $T_\infty(A)$, is the length of the longest path in the DAG. It will generally be clear which DAG we are referring to, so will drop the function notation and typically write simply T_1 and T_∞. More details on work and span can be found in [51, Ch. 27].

The DAG model specifies only the logical dependencies between strands in a computation; it does not specify the exact order in which instructions are executed, i.e. how strands are mapped to cores. Even without specifying a schedule, work and span provide us with two lower bounds on the total execution time of a DAG with work T_1 and span T_∞. Since the span is a chain of dependencies, any execution must take T_∞ time: $T_P \geq T_\infty$, where T_P is the execution time on P cores. This is known as the Span Law. On the other hand, each core can only execute one instruction at a time (in our simple performance model), so $T_P \geq T_1/P$, known as the Work Law.

The **speedup** of a computation represents the benefit from P-core execution compared to serial execution — T_1/T_P. From the Work Law above we know that $T_1/T_P \leq P$; applications

which exhibit a speedup of P achieve **linear speedup**. Although **superlinear speedup**, $T_1/T_P > P$ is impossible in our model, this may occur in practice due to memory hierarchy effects. Another bound on speedup comes from the Span Law: $T_P \geq T_\infty \implies T_1/T_P \leq T_1/T_\infty$. We call this last term the **parallelism** of a DAG, which intuitively represents the maximum speedup possible on any number of cores. It can also be thought of as the maximum number of cores that can be used to attain perfect linear speedup — once $P > T_1/T_\infty$, the second speedup bound tells us that $T_1/T_P \leq T_1/T_\infty < P$, so speedup cannot be linear. In practice, speedup is reduced by the overhead of scheduling computations, so achieving good speedup often requires parallelism to be much higher than the number of cores.

2.3 Scheduling Dynamically Multithreaded Computations

The convenient feature about the computation DAG is that it models the control-flow constraints within the program without capturing the specific choices made by the scheduler. The actual execution time of a parallel program, however, depends on which nodes are chosen to execute on each core during each time step. This is the responsibility of the scheduler. As mentioned in the previous section, the DAG unfolds dynamically, hence the scheduler must make online decisions about scheduling. In particular, at each time step the scheduler decides which **ready nodes** — those unexecuted nodes whose predecessors have all been executed — will be executed on which cores.

Naturally, it is a waste of time if there are idle cores in a system yet work to be done. Schedulers that do not allow this situation are called **greedy** [35, 64]. Ignoring scheduling overhead and assuming an ideal computer, schedulers with this property execute a computation DAG in time $T_P \leq T_1/P + T_\infty$, thus achieving linear speedup if $T_1/T_\infty \gg P$.

Building a greedy scheduler can be done by keeping a centralized pool of strands, into which cores put extra work (at a spawn strand) and from which cores take work. However, doing so leads to contention at this centralized site, resulting in high overheads in practice.

A randomized work-stealing scheduler [29], by contrast, is a decentralized scheduler, yet achieves the same theoretical performance as a greedy scheduler. The tools in this dissertation all depend on modifications to a work-stealing runtime scheduler, so we will now take some time to discuss the rules of work-stealing scheduling.

During execution, a work-stealing scheduler dynamically load balances a parallel computation across operating-system threads called **workers**, typically with one worker for each core in the system. Each worker maintains a **deque**, double-ended queue; when a worker creates nodes (e.g. by spawning a function), they are placed on the bottom of this worker's deque. When it completes its current node (i.e. returns, it takes work from the bottom of the deque. If its deque becomes empty, the worker turns into a **thief** and chooses a **victim** worker to **steal** from. Victims are chosen uniformly at random, and stolen work is always taken from the top of the victim's deque. In other words, workers treat the bottom of their own deque as a stack and the top of other deques as a queue. Throughout this dissertation steals take only a single unit of work per steal, though the work-stealing literature sometimes considers other strategies [21, 129, 134, 158].

Given a computation with work T_1 and span T_∞, a randomized work-stealing scheduler executes the computation in expected time $\frac{T_1}{P} + O(T_\infty)$ on P processors [29]. This bound can also be extended to a high probability bound. Note that this bound is asymptotically equivalent to the greedy scheduling bound and asymptotically optimal since it is at most twice the Work Law and the Span Law. Work-stealing is also effective in practice since

communication, and hence contention, only needs to occur when a worker runs out of work. For many computations steals are infrequent, keeping overheads low.

The runtime schedulers in this dissertation all serve different purposes, but they have one common feature: they allow workers to have more than a single deque. Deciding when these deques are created and how they are used allows our tools to function and determine their performance guarantees. Each of the tools was implemented by modifying the Cilk Plus [99] work-stealing scheduler to add the necessary functionality.

2.4 Expressing Parallel Programs: Fork/Join Parallelism

For most of this dissertation we will restrict our parallelism to the **fork/join** programming model, perhaps the most common abstraction for concurrency platforms. This model is often thought of as roughly corresponding to a parallel version of the divide and conquer paradigm.

Here we will use Cilk programming keywords, `spawn` and `sync`[2], to explain the fork/join programming model; other languages may differ in keywords or may provide this abstraction in other ways, such as library routines. Parallelism is created using `spawn`. When a function instance F **spawns** another function instance G by preceding the invocation with `spawn`, the **continuation** of F — the statements after the spawning of G — may execute in parallel with G without waiting for G to return. The `sync` instruction acts as a local barrier; the control flow cannot move past a `sync` in function F until functions previously spawned by F have returned. Figure 2.1 shows a simple fork/join program to compute the nth Fibonacci number.

[2]The Cilk Plus dialect actually uses `cilk_spawn` and `cilk_sync`, but we will omit the `cilk` prefix throughout this dissertation.

```
1   int fib(int n) {
2     if (n < 2) {
3       return n;
4     }
5
6     int x = spawn fib(n - 1);
7     int y = /*spawn*/ fib(n - 2);
8     sync;
9     return x + y;
10  }
```

Figure 2.1: A fork/join program using spawn and sync. Note that the second spawn is optional, since there is no code between it and the corresponding `sync`.

The Cilk Plus dialect of Cilk also provides a mechanism for parallelizing `for` loops in the form of a `cilk_for` keyword. However, this does not actually provide any power beyond the `spawn` and `sync` keywords, as these are implemented using divide-and-conquer recursion using `spawn` and `sync`.

It is important to note that these keywords denote the *logical parallelism* of the computation, not than the actual parallel execution. The runtime scheduler is responsible for deciding which logically parallel pieces of code actually execute in parallel. This frees the programmer from any kind of scheduling concerns. In fact, the programmer need not even be aware of the number of cores on the system that is to run the program. In other words, this model is **processor-oblivious**.

Fork/join programs generate a class of DAGs called **series-parallel** DAGs. These DAGs have a single **source** node (no incoming edges), a single **sink** node (no out-going edges), and can be constructed recursively:

- **Base Case**: the DAG consists of a single node that is both the source and the sink.

- **Series Composition**: let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be SP DAGs on distinct vertices. Then the graph G is formed by taking the union of the two graphs, with one additional edge from $\mathit{sink}(G_1)$ to $\mathit{source}(G_2)$, is also series parallel. Moreover, G has source and sink $\mathit{source}(G) = \mathit{source}(G_1)$ and $\mathit{sink}(G) = \mathit{sink}(G_2)$.

- **Parallel Composition**: let $G_L = (V_L, E_L)$ and $G_R = (V_R, E_R)$ be SP-dags on distinct vertices. Then the following graph is also series parallel: the graph G formed by the union of G_L, G_R, a spawn node f with edges from f to both sources, and a sync node j with edges from both sinks to j. $\mathit{source}(G) = f$ and $\mathit{sink}(G) = j$. We refer to G_L and G_R as the **left subdag** and **right subdag**, respectively, of both the spawn f and sync j.

Non-series-parallel DAGs cannot be generated by fork/join programs, so not all parallel programs can be written with this linguistic model. Many important programs, however, are expressible in this way. Moreover, the structure of series-parallel DAGs admits a space-efficient work-stealing scheduler implementation [30] and is relatively easy to reason about. Series-parallel structure is vital to the analysis of the systems in chapters 3 and 4.

2.5 Determinism and Races

One of the most challenging aspects of parallel programming is dealing with shared memory. A dynamically multithreaded program in which every memory location is updated with the same sequence of values in every execution on a given input is deterministic. Deterministic programs will always produce the same DAG given the same input, and will behave the same no matter how they are scheduled. Non-deterministic programs, on the other hand, are

the bane of parallel programmers, since bugs may appear and disappear based on how the program is scheduled.

In a parallel program intended to be deterministic, a **determinacy race** [71] (or **general race** [137]) occurs when multiple logically parallel instructions access the same memory location, and at least one is a write. Determinacy races can cause non-deterministic behavior and are often bugs. Worse, because these bugs rely on the order of concurrent instructions, they may lie dormant for a long period before a particular sequence causes a malfunction. For example, the massive 2003 power blackout in the Northeastern US was caused, in part, by a race condition [151].

Determinacy races have called various names in the literature, including harmful shared-memory access [140], race conditions [172], access anomalies [61], and data races [132]. We have taken our definition of determinacy race from Netzer and Miller [137] who formally define and discuss various types of races. Chapters 4, 7 and 8 present our work on automatically detecting these types of races at runtime. We discuss a different debugging method for a different type of non-determinism bug, called an atomicity bug, in chapter 6.

Chapter 3

Implicitly Batching Data Structure Operations with Batcher

Many algorithms rely on having efficient data structures, allowing large amounts of data to be stored, accessed, and modified efficiently. When used sequentially, they allow algorithm designers to reason separately about the algorithm itself and data structure operations. This modularity allows programmers to easy analyze the runtime of algorithms that use data structures.

A common approach when using data structures within parallel programs is to employ **concurrent data structures** — data structures that can cope with multiple simultaneous accesses. Designing and using concurrent data structures, however, is challenging. In fact, the design of a new concurrent data structure is often a publishable result. Moreover, the existing performance theorems do not often imply linear speedup for the enclosing program — the normal way to analyze parallel algorithms using data structures is to assume the worst-case latency for each operation. In most cases the worst-case is linear in the number

of processors, $\Omega(P)$. Assuming n operations, the running time is at least $\Omega(nP/P) = \Omega(n)$ — the same as the sequential runtime!

In some sense concurrent data structures make the problem harder than it needs to be by trying to cope with arbitrary access patterns. They make no use of the fact that all data structure operations belong to the same enclosing program. A key idea in this chapter is to leverage the fact that operations can in fact coordinate with each other with the help of a carefully designed runtime scheduler.

An alternative to concurrent data structures allows for a manual version of such coordination: batched data structures. These data structures are designed to operate on a group of data structure operations collectively. The programmer is responsible for manually collecting a group of operations and for ensuring no other operations are invoked while a "batch" is in progress. These structures are easier to design than concurrent data structures since only one batch is active at a time, allowing parallelism in batches while reducing the need for complicated concurrency control. They are also relatively easy to analyze – we can simply add the running time of a sequence of batches to the enclosing program. For example, the running time of parallel algorithms for shortest paths and minimum spanning tree [36, 62, 145] have been proved using batched parallel priority queues [36, 53, 62, 163].

Unfortunately, applying this technique to existing algorithms often requires severe code restructuring. In some cases a restructuring is actually impossible. Consider the case of an on-the-fly race detector [132, 19, 154], which updates order-maintenance data structures on spawns and syncs in a program. To correctly detect races the data structure must be updated before continuing – delaying operations could miss some races. So it seems impossible to reorganize operations into batches in this case. We will consider this case further in chapter 4.

This chapter describes Batcher [4], which does this kind of "batching" coordination automatically and behind the scenes. The scheduler we design allows batched data structures to be used with a broad class of parallel programs that make parallel accesses to data structures. The performance theorem we prove allows data structures to be analyzed, meaning that a parallel program may be analyzed without considering the specific data structure implementation. The performance theorem also achieves an efficient running time with the help of a special runtime scheduler. For example, for n access to a search tree with large enough n, our theorem yields a completion time of $\Theta(n \lg n/P)$, which is asymptotically optimal and has linear speedup. We are unaware of any comparable aggregate bounds for concurrent search trees.

This technique of **implicit batching** essentially achieves the benefits of both concurrent and batched data structures. The programmer provides two components: (1) a parallel program $\mathcal{C}$ containing parallel accesses to a abstract data type (ADT) $\mathcal{D}$, and (2) a batched data structure implementing the data structure $\mathcal{D}$. The scheduler dynamically and transparently organizes the program's parallel accesses to the data structure into batches, with at most one batch executing at a time.

Using implicit batching gives the benefits of batched data structures without restructuring the enclosing program $\mathcal{C}$. The scheduler is responsible for grouping any concurrent accesses to the abstract data type $\mathcal{D}$ into batches and invoking the appropriate implementation of a batched operation. The programmer need only implement this batched operation, which may be implemented using dynamic multithreading. Since our scheduler handles all synchronization and ensures that at most one batch is executed at a time, such batch operations can typically avoid handling concurrency, omitting locks or atomic operations.

Implicit batching closely resembles flat combining [88], where each concurrent access to a data structure queues up in a list of operation records, and this list of records (i.e., a batch) is later executed sequentially. Implicit batching may be viewed as a generalization of flat combining in that it allows *parallel* implementations of batched operations, instead of only a sequential one allowed by flat combining. Due to sequential batches, flat combining does not guarantee provable speedup guarantees. However, flat combining has been shown to be more efficient in practice than some of the best concurrent data structures under certain loads. We implemented a prototype of Batcher showing that it has the potential to improve performance even more than flat combining.

The rest of this chapter is structured as follows. First we discuss Batcher's interface and some example data structures in section 3.1. This is followed by a discussion of the extensions to the work stealing runtime scheduler required by Batcher in section 3.2. Next section 3.3 provides a theoretical analysis of batched data structures that use Batcher. Section 3.4 presents a short evaluation of our prototype implementation, while section 3.5 discusses related work. Finally, we provide some concluding thoughts and potential future work in section 3.6.

3.1 Using Batcher

We first describe how to use Batcher— designing and analyzing data structures — without any knowledge of the runtime system; the scheduler design is deferred to section 3.2. We distinguish between two different types of programmers. In the sequential case the data structure programmer implements an ADT $\mathcal{D}$, while the algorithm programmer writes a program $\mathcal{C}$ that makes calls to this ADT's interface. With Batcher each of these programmers instead interface with the Batcher runtime system, which combines these two modules and performs scheduling.

Wherever the program would make a call to the ADT interface, it instead makes a call into the runtime system using the BATCHIFY operation. This function takes in a pointer to the data structure and a pointer to an **operation record**. The operation record contains data about the required operation (e.g. an element to insert into the data structure), as well as a location into which to place the result. As far as the algorithm programmer is concerned, BATCHIFY resembles a normal procedure call to access a *concurrent data structure*, and the control flow blocks at this point until the operation completes. Figures 3.1 and 3.2 shows how a simple shared counter can be implemented using Batcher. We will discuss this example further in section 3.1.2.

Instead of providing an interface to the algorithm programmer, the data structure programmer provides a batched operation, denoted by BOP. A batched operations receives an array of operation records with information about the necessary operations. Since Batcher guarantees that at most one batch operation is in progress at a time, the batch operation need not consider any concurrent accesses from other operations. This simple access pattern makes it easier to design parallel batch operations – these operations can generate parallelism like any dynamically multithreaded program, e.g. using `spawn`/`sync` or parallel loops.

3.1.1 Analyzing Programs with Batcher

One of the promises of Batcher is the ability to analyze algorithms and data structures separately and combine the results easily. Hence we need a separate model for algorithms and batched data structures, even though the two unfold together as part of a computation.

Recall from chapter 2 that we can model dynamic multithreaded computations as DAGs, and that at each time-step the scheduler decides which nodes to execute. We model the enclosing program as such a DAG except that we replace calls to BATCHIFY with special

data structure nodes which may take longer than unit-time to execute. We call this structure the **core DAG**, defining the **core work** T_1 as the number of nodes in the core DAG, and the **core span** T_∞ as the longest path through the core DAG in terms of number of nodes. This is just as in chapter 2, ignoring the fact that data structure nodes are not unit-time. We let n denote the total number of data structure nodes and m denote the maximum data structure nodes along any path in the DAG. Surprisingly, the analysis of our scheduler does not need any information about these nodes.

Each call A to a batched data structure operation (i.e. an invocation of BOP) is modeled as a separate subcomputation with a DAG G_A. Accordingly, we let w_A and s_A denote the work and span of this batch DAG. Additionally, $w_P(n)$ is defined to be the maximum total work for any sequences of batches which execute n total data structure operations (n calls to BATCHIFY). We define the **data structure span** $s_P(n)$ as the worst-case span of any batch DAG G_A in any such sequence subject to the restriction that $w_A/s_A = O(P)$, meaning that the batch has limited parallelism. When the data structure's analysis is not amortized this definition can be simplified to the worst-case span of any size-P batch DAG. We assume a fixed P, so we will use $w(n)$ and $s(n)$ as shorthand for $w_P(n)$ and $s_P(n)$ throughout this chapter.

With that analytic model, in section 3.3 we prove that Batcher provides the following performance guarantee:

Theorem 3.1. *Batcher executes such a program in expected time at most*

$$O\left(\frac{T_1}{P} + T_\infty + \frac{w(n) + ns(n)}{P} + ms(n)\right) .$$

assuming that $s(n) \geq \lg P$ *(true since we only allow binary forking) and that the only synchronization of the program occurs through* `syncs`*; the algorithm or data structure code does not use explicit synchronization primitives such as locks or atomic operations.*

Note that $w(n)$ and $s(n)$ are metrics of the data structure itself — they do not consider how an enclosing program uses the data structure! Hence Batcher makes using data structures in parallel programs nearly as easy as in sequential programs — you can analyze the data structure separately from the algorithm that uses it. As we will see in the next section, this theorem implies nearly linear speedup for many parallel batched data structures.

3.1.2 Batched Data Structure Examples

To illustrate the use of Batcher and its performance bound, we now present some examples. We are not claiming any new data structure ideas here, we only want to drive home the power of batched data structures and show how to apply the bound.

Concurrent counter. As a simple example, consider a core program that makes n completely parallel increments to a shared counter, as given by figure 3.1. This example is for illustration only, and is not intended to be very deep. This program has $\Theta(n)$ core work and $\Theta(\lg n)$ core span (with binary forking). The shared counter is an abstract data type that supports a single operation INCREMENT, which atomically adds a value (possibly negative) to the counter and returns its current value.

A trivial concurrent counter uses atomic primitives like fetch-and-add to INCREMENT. If the primitive is mutually exclusive (which is true for fetch-and-add on current hardware), then n INCREMENTs take $\Omega(n)$ time. The total running time of the program is thus $\Omega(n)$ regardless of the number of processors.

```
1  parallel_for i = 1 to n
2        B[i] = INCREMENT(A[i])
```

Figure 3.1: A parallel loop that performs n parallel updates to a shared counter. Here, $A[1\,..\,n]$ is an array of values by which to increment (or decrement if negative) the counter, and $B[1\,..\,n]$ holds any return values from the INCREMENTs.

```
3  struct OpRecord {int value; int result;}

INCREMENT(int x)
4  OpRecord op
5  op.value = x
6  BATCHIFY(this, op) //ask the scheduler to batch op
7  return op.result

BOP(OpRecord D[1..size])
 8  let v be the value of the counter
 9  D[1]'s value field = v + D[1]'s value
10  perform parallel-prefix-sums on value fields of D[1..size],
                                    storing sums into result fields of D[1..size]
    // now D[i]'s result = sum_{k=1}^{i} D[k]'s value
11  set the counter to D[size]'s result
```

Figure 3.2: A batched-counter implementation. As we shall see in section 3.2, line 6 *logically* blocks, but the processor does not spin-wait. The BOP is called by the scheduler automatically.

One could instead use a provably efficient concurrent counter, e.g., by using the more complicated combining funnels [168, 167]. Doing so would indeed yield a good overall running time, but these techniques are not applicable to more general data structures. As we shall see next, the implicitly batched counter achieves good asymptotic speedup with a trivial implementation.

Figure 3.2 shows a sample batched counter. Here, when the core program makes an INCREMENT call, it creates an operation record which is handed-off to the scheduler. The scheduler

later runs the batch increment BOP on a set of increments. The main subroutine of the batched operation is "parallel prefix sums", which *in parallel* computes $\sum_{k=1}^{i} D[k]$ for every i. It is easy to prove that returning $\sum_{k=1}^{i} D[k]$ yields linearizable [91] counter operations. Prefix sums is a commonly used and powerful primitive in parallel algorithms, and hence we consider this 4-line implementation of BOP to be trivial. Adaptations of Ladner and Fischer's approach to prefix sums [114] to the fork-join model have $O(x)$ work and $(\lg x)$ span for x elements.

To analyze the execution of this program using Batcher, we need only bound $w(n)$, the total work of arbitrarily batching n operations, and $s(n)$, the span of a batched operation that processes P operation records (performs P increment operations). Since the work of prefix sums is linear, we have $w(n) = \Theta(n)$. Since a size-P batch has $O(\lg P)$ span (dominated by prefix sums), we have $s(n) = O(\lg P)$. We thus get the bound $O(\frac{T_1 + n \lg P}{P} + m \lg P + T_\infty)$ for performing n INCREMENTs, with at most m along any path. The core dag of figure 3.1 has $T_1 = O(n)$, $T_\infty = O(\lg n)$, $m = 1$, so we have a running time of $O(\frac{n \lg P}{P} + \lg n)$ for $n > P$. This nearly linear speedup is much better than for the trivial counter.

Search tree. There exists an efficient batched 2-3 tree [148] in the PRAM model, and it is not too hard to adapt this algorithm to dynamic multithreading. The main challenge in a search tree is when all inserts occur in the same node of the tree, e.g., when inserting P identical keys. The main idea of this batched search tree is to first sort the new elements, then insert the middle element and recurse on each half of the remaining elements. This process allows for each of the new keys to be separated by existing keys without concurrency control. It is not obvious how to leverage the same idea in a concurrent search tree.

See [148] for details of the batched search tree. Suffice it to say that a size-x batch is dominated by two steps: (1) a parallel search for the location of each key in the tree, having $O(x \lg n)$ work and $O(\lg n + \lg x)$ span, and (2) a parallel sort of the x keys, having $O(x \lg x)$ work. The data structure span is thus $s(n) = O(\lg n + sort(P))$, where $sort(P) = O(\lg P \lg\lg P)$ [50] is the span of a parallel sort on P elements in the dynamic-multithreading model. The data structure work $w(n)$ is maximized for n/P batches of size $x = P$, yielding $w(n) = O(n \lg n)$ data structure work. Applying theorem 3.1, we get a running time of $O(\frac{T_1+w(n)+s(n)}{P} + ms(n) + T_\infty) = O(\frac{T_1+n\lg n+n\lg P\lg\lg P}{P} + m\lg n + m\lg P\lg\lg P + T_\infty)$. For large enough n (specifically, $n = \Omega(P^{\lg\lg P})$, this reduces to $O(\frac{T_1+n\lg n}{P} + m\lg n + T_\infty)$, which is asymptotically optimal in the comparison model and provides linear speedup for programs with sufficient parallelism. For instance, a program obtained by substituting the increment operation with an insert in figure 3.1 would yield the running time of $O(n \lg n/P)$, implying linear speedup, even though the program only performs data structure accesses.

Amortized stack. We now briefly describe an example, namely a LIFO stack, which has amortized performance bounds. The data structure is an array that supports two operations: a PUSH that inserts an element at the end of the array, and a POP that removes and returns the last element. Such an array can be implemented using a standard table doubling [51] technique, whereby the underlying table is rebuilt (in parallel) whenever it becomes too full or too empty. To PUSH a batch of x elements into an n-element array, check if $n+x$ elements fit in the current array. If so, in parallel simply insert the ith batch element into the $(n+i)$th slot of the array. If not, first resize the array by allocating new space and copying all existing elements in parallel. POPs can be simultaneously supported by breaking the batch into a PUSH phase followed by a POP phase.

To analyze this data structure, the (amortized) work of a size-x batch is $\Theta(x)$, yielding $w(n) = \Theta(n)$ (worst case). The work of any individual batch, however, can be as high as $\Theta(n)$ when a table doubling occurs. More importantly, any batch A that has w_A batch work has batch span $s_A = O(\lg w_A)$. Hence any batch A that performs $w_A \geq P^2$ work has parallelism $w_A/s_A = \Omega(P^2/\lg P)$. We thus conclude that the data structure span is $s(n) = O(\lg(P^2)) = O(\lg P)$. Plugging these bounds into theorem 3.1, we get a total running time of $O(\frac{T_1+n\lg P}{P} + m\lg P + T_\infty)$.

3.2 Runtime Scheduler

This section presents the high-level design of the Batcher scheduler, a variant of a distributed work-stealing scheduler. Since Batcher is a distributed scheduler, there is no centralized scheduler thread and the operation of the scheduler can be described in terms of state-transition rules followed by each of the P workers. First, we overview some important properties of Batcher and the intuition for its performance analysis. Then we describe the internal state that Batcher maintains in order to implicitly batch data structure operations and to coordinate between executing the core and batch dags. We then describe how batches are launched and how load-balancing is done using work-stealing.

At a high level, calls to BATCHIFY correspond to data structure nodes and Batcher is responsible for implicitly batching these data structure operations and then executing these batches by calling BOP. When a worker p encounters a data structure node u (i.e., p executes a call to BATCHIFY), p alerts the scheduler to the operation by creating an operation record *op* for that operation and placing it in a particular memory location reserved for this processor. Eventually, *op* will be part of some batch A and the scheduler will call BOP on A. Unlike core nodes, however, the data structure node can logically block for longer than

one time step and u's successor(s) in the dag do not become ready until after this call to BOP returns, that is, the operation corresponding to u is actually performed on the data structure as part of a batch.

Inherent to implicit batching is the idea that the batch the scheduler invokes only one batch at a time. Hence the data-structure implementation need not cope with concurrency, simplifying the data-structure design. The following invariant states this property for Batcher.

Invariant 3.2. At any time during a Batcher execution, at most one batch is executing.

There are many choices that go into a scheduler for implicit batching. For Batcher, we made specific choices guided by the goal of proving a performance theorem. Three of the main questions are what basic type of scheduler to use, how large batches should be, and when and how batches are launched. As far as the low-level details are concerned, we chose in favor of simplicity where possible. Batcher restricts batch sizes, as stated by the following invariant; this size cap ameliorates application of the main theorem as it simplifies the analysis of any specific data structure.

Invariant 3.3. In a Batcher execution, batches contain at most P data structure nodes.

Finally, whenever an operation record is created and no batch is currently in progress, Batcher immediately launches a new batch; it does not wait for a certain number of operations to accrue; this decision is important for the theoretical analysis. Therefore, batches can contain as few as one operation. Launching a batch includes some (parallel) setup to gather all operation outstanding operation records, executing the provided (parallel) batched operation BOP thereby inducing a batch dag, and some (parallel) cleanup after completing. Since the setup/cleanup overhead is scheduler dependent, we account for the overhead separately, and the batch dag comprises only the steps of BOP.

Intuition behind the analysis. We have already exposed one significant difficulty for analysis of Batcher: since batches launch as soon as possible, some batches may contain just a single data-structure node. If this were true for every batch, then all operations would be sequentialized according to invariant 3.2, and it would seem impossible to show good speedup. In addition, the batch setup and cleanup overhead is the same, regardless of batch size; therefore, having many small batches may incur significant overhead.

Fortunately, small batches fall into two cases, both being good: (1) Many data structure nodes accrue while a small batch is executing. These will be part of the next batch, meaning that the next batch will be large and make progress toward the batch work $w(n)$. (2) Not many data structure nodes are accruing. Then the core dag is not blocked on too many data-structure nodes, and progress is being made on the core work T_1. In both cases, the setup and cleanup overhead of the small batch can be amortized either against the work done in the next batch or the work done in the core dag.

3.2.1 Batcher State

The Batcher scheduler maintains three categories of shared state: (1) collections for tracking the implicitly batched data structure operations, (2) status flags for synchronizing the scheduler, and (3) deques for each worker tracking execution-DAG nodes and used by work stealing. With the exception of one global flag, most of this state is distributed across workers, with each worker only managing specific updates according to the provided rules that define the scheduler.

To track active data structure nodes, Batcher maintains two arrays. When a worker encounters a data structure node (executes a call to BATCHIFY(*op*)), instead of accessing the data structure directly, an operation record *op* is created and placed in the **pending array**

and the data structure node is suspended. Batcher guarantees that each worker has at most one suspended node / pending operation at any time; therefore this pending array may be maintained as a size-P array, with a dedicated slot for each of the P workers. Batcher also maintains a **working set**, which is a densely packed array of all the operation records being processed as part of the currently executing batch.

To synchronize batch executions, Batcher maintains a single global **active-batch** flag. In addition, each worker p has a local **work-status** flag (denoted $Status[p]$), which describes the status of p's current data structure node. Batcher guarantees that at any instant, each worker has at most one data structure node u that it is trying to execute. For concreteness, think in terms of the following four states for worker status $Status[p]$:

- `pending`, if p has an operation record op for a suspended data structure node u in the pending array.
- `executing`, if p has an operation record op for a suspended data structure node u in the working set, i.e., a batch containing u is currently executing.
- `done`, if the batch A containing u has completed its computation, but p has not yet resumed the suspended node u.
- `free`, if p has no suspended data structure node.

If $Status[p]$ is `pending`, `executing`, or `done`, we say p is **trapped** on operation u. Otherwise, we say p is **free**.

Finally, Batcher maintains two deques of ready nodes on each worker: a **core deque** for ready nodes from the core dag, and a **batch deque** for ready nodes from a batch dag. In particular, the deques in Batcher obey the following invariant:

Invariant 3.4. Ready nodes belonging to the core dag G are always placed on some worker's core deque, whereas ready nodes that belong to some batch A's batch dag G_A are always placed on some worker's batch deque.

Associated with these deques, each worker p also has an **assigned node** — the node that p is currently executing. At any instant, the assigned node of p may conceptually be associated with either the core deque or the batch deque, depending on the type of node being executed by that worker. Some workers may be executing core nodes while others are executing batch nodes.

3.2.2 Batcher Algorithm

Batcher uses a variant of work stealing, with some augmentations to support implicit batching. Free workers and trapped workers behave quite differently. Initially all workers are free, and all ready nodes belong to the core dag, and Batcher behaves similarly to traditional work stealing. As data structure nodes are encountered, however, the situation changes. The scheduling rules are outlined in figure 3.3 and described below.

Free workers behave closest to traditional work stealing. A free worker is allowed to execute any node (core or batch), but it only steals if both of its deques are empty. Specifically, if either deque is nonempty, the worker executes a node off the nonempty deque, and any newly enabled nodes are placed on the same deque. Batcher thus maintains the following invariant:

Invariant 3.5. Workers that have `free` status in Batcher can have at most one of their deques non-empty, i.e., they have nodes either on the batch deque or core deque, but not both.

When p is free and both deques are empty:
    steal from random victim, using alternating-steal policy

When a data-structure node u is assigned to (free) worker p:
    insert operation record into *pending*[p]
    Status[p] = `pending`
    suspend u
    // p is now trapped

When p is trapped and its batch deque is empty:
    if *Status*[p] = `done`
        Status[p] = `free`
        resume executing the core deque from
                                suspended data-structure node u
        // p is now free
    else if global batch flag = 0 and
                        compare-and-swap(global batch flag, 0, 1)
            run LaunchBatch
        else steal from random victim's batch deque

Figure 3.3: Scheduler-state transition rules invoked by workers with empty deques. When the appropriate deque is not empty, the worker removes the bottom node from the deque and executes it.

If both deques are empty, however, the free worker performs a steal attempt according to an **alternating-steal policy**: each worker's kth steal attempt (successful or not) is from a random victim's core deque if k is even, and from a random victim's batch deque if k is odd. The alternating-steal policy is important to achieve the performance bound in section 3.3.

When a (free) worker p executes a data structure node u, p first inserts the corresponding operation record op in its dedicated slot in the pending array, and then it changes its own status to `pending`. At this point, the p becomes trapped on u, and according to invariant 3.5, it has an initially empty batch deque.

Unlike free workers, which are allowed to execute both core and batch work, trapped workers are only allowed to execute nodes from a batch deque. If a trapped worker p has a nonempty batch deque, it simply selects a node off the batch deque as in traditional work stealing. If it has an empty batch deque, however, it performs the following step. First, it checks whether its data structure node u has finished, i.e., if $Status[p] =$ `done`. If so, it changes its own status to `free` and resumes from the suspended data structure node on the core deque. Otherwise, it checks the global batch status flag and tries to set it using an atomic operation if no batch is executing. If successful in setting the flag, p "launches" a batch. If it is unsuccessful (someone else successfully set the flag and launched a batch) or if a batch is already executing (status flag was already 1) it simply tries to steal from a random victim's deque. Batcher guarantees that if no batch is executing, then all workers have status either `pending`, `done` or `free`; therefore, only pending workers can succeed in launching a new batch.

Launching a batch. Launching a batch corresponds to injecting the parallel task LAUNCH-BATCH (see figure 3.4), i.e., by inserting the root of the subdag induced by this code on a worker's batch deque. This process has five steps. First, the pending array is processed in

```
LaunchBatch()
1  parallel_for i = 1 to P
       if Status[i] = pending then Status[i] = executing
2  compact all executing op records, moving them from
                                              pending array to working set
       // using parallel prefix sums subroutine
3  execute Bop (the actual parallel batch) on records in working set
4  parallel_for i = 1 to P
       if Status[i] = executing then Status[i] = done
       remove done op records from the working set.
5  reset global batch-status flag to 0
```

Figure 3.4: Pseudocode for launching a batch. This method executes as an ordinary task in a dynamic multithreaded computation, i.e., it may run using any number of workers between 1 to P workers, depending on how work-stealing occurs.

parallel, changing the status of all `pending` workers to `executing`, thereby acknowledging the operation record. Second, the `executing` records are packed together in the working-set array, which can be performed in parallel using a parallel prefix sums computation. Third, the actual batched operation (Bop) is executed on the records in working set. Fourth, the pending array is again processed in parallel, changing the status of all `executing` workers to `done`. Finally, the batch-status flag is reset to 0. In practice, several of these steps can be merged, but we are not concerned about these low-level optimizations in this paper.

As mentioned above, launching a batch incurs some overhead, such as updating status fields and compacting the pending array into working-set, beyond the execution of the batched operation Bop itself. We refer to this overhead as the **batch-setup overhead**. Note that this set-up procedure is itself a dynamic multithreaded program with $\Theta(P)$ work and $\Theta(\lg P)$ span, primarily due to the cost of the **parallel_for** and parallel prefix sums computations over P elements. This set-up work is performed in exactly the same way as the batched operation Bop is performed — that is, the nodes of this procedure are placed on batch

deques and are executed in parallel (via work-stealing) by workers working on these deques. Note, however, that for the purposes of the dag metrics, the overhead is *not* counted as part of the batch work, batch span, or data structure span; this omission is by design since the overhead is a function of the scheduler, not the input program. This fact is one of the challenges in proving that Batcher has a good running time, and it is exacerbated by the fact that the overhead is as high for a batch containing 1 operation as it is for a batch containing P operations. Nevertheless, we shall show (section 3.3) that Batcher is provably efficient.

Not trapped long. The following lemma shows that a worker is not trapped for very long by a particular operation (at most 2 batches).

Lemma 3.6. *Once the operation record for a data structure node u is put into the pending array, at most two batches execute before the node completes.*

Proof. Consider an operation u whose status changes at time t to `pending`. Any batch finishing before time t does not delay u. Any batch A launched after time t observes u in the pending array, and incorporates it in A and completes it; this accounts for one batch execution. According to invariant 3.2, there can only be one batch that is launched before t and finishes after t, which accounts for the second batch. □

Correctness of state changes. Each worker is responsible for changing its own state from `done` to `free` and `free` to `pending`. There is thus no risk of any races on these state changes. The changes from `pending` to `executing` and `executing` to `done` may be performed by an arbitrary worker, but these changes occur as part of the parallel computation LaunchBatch. Since LaunchBatch is itself race free and only one LaunchBatch occurs at a time (protected by the global batch-status flag), these transitions are also safe.

3.3 Theoretical Analysis

We now analyze the performance of Batcher. We first provide some definitions and the statement of the completion time bounds. Then we use a potential function argument to prove the performance bounds.

Definitions and theorem statements. We will analyze the running time using the computation dag G and the set of batch dags that represent batches generated due to implicit batching performed by Batcher. We will analyze the running time using an arbitrary parameter τ, which will be later related to the data structure span $s(n)$. We define a few different types of batches. A batch A is **τ-wide** if its batch work is more than $P\tau$. A batch is **τ-long** if its batch span is more than τ. These definitions only count the work and span within the batched operations themselves, not the batch-setup overhead due to Batcher. Since τ is implied, we often drop it and call batches wide or long. Finally, a batch is **popular** if it processes more than $P/4$ operation records; that is, it contains more than $P/4$ data structure nodes. A batch is **big** if it is either long, wide or popular, or if it occurs immediately before or after a long, wide or popular batch. All other batches are **small**.

The above definitions are with respect to individual batches that arise during an execution. We next define a property of the data structure itself, analogous to data structure work.

Definition 3.7. Consider any sequence of parallel batched operations and a real value τ. The **τ-trimmed span of the sequence** of batches is the sum of the spans of the *long* batches in the sequence. The **τ-trimmed span of a data structure**, denoted by $S_\tau(n)$, is the worst-case τ-trimmed span for n data structure nodes grouped arbitrarily into batches.

We now state our main theorem, a bound on the total running time of a Batcher computation, which is proven at the end of this section. The restriction that $\tau \geq \lg P$ arises from binary forking.

Theorem 3.8. *Consider a computation with T_1 core work, T_∞ core span, and n data structure nodes with at most m falling along any path through the dag. For any $\tau \geq \lg P$, let $S_\tau(n)$ and $w(n)$ denote the worst-case τ-trimmed span and total work of the data structure, respectively. Then Batcher executes the program in $O\left(\frac{T_1+w(n)+n\tau}{P} + T_\infty + S_\tau(n) + m\tau\right)$ expected time on P processors.*

This theorem holds for any $\tau \geq \lg P$; however, it does not provide intuition about which τ is best. There is a trade-off: increasing τ increases $n\tau$ and $m\tau$, but decreasing τ increases $S_\tau(n)$ since more batches become long. As we shall see at the end of this section, setting $\tau = s(n)$, the data structure span, yields theorem 3.1 as a corollary since other terms dominate $S_\tau(n)$.

Corollary 3.9. *Batcher executes the program described in theorem 3.8 in expected time*

$$O\left(\frac{T_1 + w(n) + ns(n)}{P} + ms(n) + T_\infty\right).$$

3.3.1 Proof Approach

As with previous work-stealing analyses, our analysis separately bounds the total number of processor steps devoted to various activities; in our case, these activities are core work, data structure work, stealing (and failed steal attempts), and the batch-setup overhead. We then divide this total by P, since each processor performs one processor step per time step, to get the completion time.

It is relatively straightforward to see that the number of processor steps devoted to core work is T_1 and the number of time steps devoted to data structure work is $w(n)$. The difficulty is in bounding the number of steal attempts and the batch set up overhead. To bound the number of steal attempts, we adopt a *potential function* argument similar to Arora et al.'s work-stealing analysis [8], henceforth referred to as ABP. In the ABP analysis, each ready node is assigned a potential that decreases geometrically with its distance from the start of the dag. For traditional work stealing, one can prove that most of the potential is in the ready nodes at the top of the deques, as these are the ones that occur earliest in the dag. Therefore, $\Theta(P)$ random steal attempts suffice to process all of these nodes on top of the deques, causing the potential to decrease significantly. Therefore, one can prove that $O(PT_\infty)$ steal attempts are sufficient to reduce the potential to 0 in expectation.

The ABP analysis does not directly apply to bounding the number of steal attempts by Batcher for the following reason. When a data structure node u becomes ready and is assigned to worker p, p places the corresponding operation record in the pending array and u remains assigned (control flow does not go past u) until results from u are available. But u may contain most of the potential of the entire computation (particularly if p's deque is empty; in this case u has all of p's potential). Since u cannot be stolen, steals are no longer effective in reducing the potential of the computation until the batch containing u completes. To cope with this difficulty, we apply different progress arguments to big batches and small batches.

Big batch steal attempts. For big batches, we apply the ABP potential function to each batch's computation dag. Nodes in a batch dag are never "suspended" in the way data structure nodes are, so the ABP argument applies nearly directly. We charge this case against the τ-trimmed span or the data structure work. (As a technical detail, we must also show

that P steal attempts overall equate to $\Omega(P)$ steal attempts from batch deques in order to complete the argument.)

Other steal attempts. Unfortunately, small batches do not contribute to the τ-trimmed span, so the above approach does not apply.[1] Instead, we apply extra machinery to bound these steal attempts. The intuition is that if many steal attempts actually occur during a small batch, then the batch should complete quickly (i.e., within $O(\tau)$ time steps). On the other hand, if few steal attempts occur then the workers are being productive anyway, since they are doing useful work (either core work or data structure work) instead of stealing. To apply this intuition more formally within the ABP framework, we augment each data structure node to comprise a chain of τ "dummy nodes," which captures these cases by appropriate potential decreases in the augmented dag.

3.3.2 DAG Augmentation and Potential Function

We create an augmented computation dag, the $\tau-$**execution dag** $G(\tau)$, by adding a length $\Theta(\tau)$ chain of **dummy nodes** before each data structure node in the computation dag. The work of this dag is $\mathcal{W}_{G(\tau)} = T_1 + O(n\tau)$ and span is $\mathcal{S}_{G(\tau)} = T_\infty + O(m\tau)$.

For the purpose of the analysis, we suppose the scheduler executes the augmented dag instead of the original dag. The scheduler operates with one corresponding difference: when a worker encounters a data structure node, this node remains assigned to the worker, but $\Theta(\tau)$ nodes of the dummy-node chain are placed at the bottom of its core deque. If a worker p steals from another worker p''s core deque and a dummy node is on the top of that deque, then p steals and immediately processes the dummy node. This steal is considered a successful

[1] Adding even P steal attempts for each of potentially n small batches would result in $\Omega(nP)$ steal attempts or $\Omega(n)$ running time, i.e., no parallelism.

steal attempt. When a worker returns from a batch, all the dummy nodes on the bottom of its deque disappear. Note that dummy nodes are only for accounting. Operationally, this runtime system is identical to the one described in section 3.2, except the analysis now just counts some unsuccessful steals as successful steals. More precisely, whenever a dummy node is stolen from a victim's deque, the corresponding steal in the real execution is unsuccessful because the victim's deque was empty.

We now define the potentials using this augmented dag. Each node in G has **depth** $d(u)$ and **weight** $w(u) = \mathcal{S}_G - d(u)$. Similarly, for a node u in the batch dag G_A, $d(u)$ is its depth in that dag, and its weight is $w(u) = s_A - d(u)$. The weights are always positive.

Definition 3.10. The **potential** Φ_u of a node u is $3^{2w(u)-1}$ if u is assigned, and $3^{2w(u)}$ if u is ready.

The **core potential** of the computation is the sum of potentials of all (ready or assigned) nodes $u \in G$. The **batch potential** is the sum of the potentials of all $u \in G_A$ where A is the currently active batch (if one exists).

The following structural lemmas follow in a straightforward manner from the arguments used throughout the ABP paper [8], so we state them without proof here. [2]

Lemma 3.11. *The initial core potential is* 3^{S_G} *and it never increases during the computation.*

Lemma 3.12. *Let* $\Phi(t)$ *denote the potential of the core dag at time* t*. If no trapped worker's deque is empty, then after* $2P$ *subsequent steal attempts from core deques the core potential is at most* $\Phi(t)/4$ *with probability at least* $1/4$*.*

[2] ABP does not explicitly capture these three lemmas as claims in their paper — some of their proof is captured by "Lemma 8" and "Theorem 9" of [8], but the rest falls to inter-proof discussion within the paper .

Lemma 3.13. *Suppose a computation (core or batch) has span S, and that every "round" decreases its potential by a constant factor with at least a constant probability. Then the computation completes after $O(S)$ rounds in expectation, and the total number of rounds is $O(S + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$.*

The following two lemmas extend lemmas 3.11 and 3.12 to batch potentials. The proofs of these lemmas can also be derived from ABP proofs in a similar manner.

Lemma 3.14. *The batch potential Φ_A increases from 0 to 3^{2s_A} when A becomes ready, and never increases thereafter.*

Lemma 3.15. *Let $\Phi_A(t)$ be the potential of batch A at time t. After $2P$ subsequent steal attempts from batch deques, the potential of A is at most $\Phi_A(t)/4$ with probability at least $1/4$.*

Since different arguments are required for big and small batches, we partition steal attempts into three categories. A **big-batch steal attempt** is any steal attempt that occurs on a time step during which a big batch is executing. A **trapped steal attempt** is a steal attempt made by a trapped worker (a worker whose status is not `free`) on a time step when no big batch is active. A **free steal attempt** is a steal attempt by a free worker (a worker whose status is `free`) on a time step when no big batch is active. We can now bound the different types of steal attempts and the batch-setup overhead.

Big-batch steal attempts. The big-batch steal attempts are bounded by the following lemma. The proof of this lemma is the most straightforward of the three cases.

Lemma 3.16. *The expected number of big-batch steal attempts is $O(n\tau + PS_\tau(n) + w(n))$.*

Proof. We first prove that if L is the set of big batches, the expected number of big batch steal attempts is $O(P \sum_{A \in L} s_A)$.

Consider a particular big batch A. When the first round starts, the potential of the batch is 3^{2s_A} (lemma 3.14). Divide the steal attempts that occur while the batch is executing into rounds of $4P$ steal attempts, except for the last round, which may have fewer. While A is executing, at least half the steal attempts are from batch deques, since all the trapped steals are from batch deques, and half the free steals are from batch deques by the alternating steal policy. Therefore, in every round, at least $2P$ steal attempts are from batch deques. Applying lemma 3.15, the potential of the batch decreases by a constant factor with probability $1/4$ during each round. Therefore, applying lemma 3.13, we can conclude that there are expected $O(s_A)$ rounds while A is active.

We use linearity of expectation to add over all big batches. We first add over long, wide and popular batches. The total span of long batches is $S_\tau(n)$ by definition. There are at most $w(n)/P\tau$ wide batches, and at most n/P popular batches. If they are not also long, they have span less than τ. We triple the number to account for batches before and after long, wide or popular batches. Therefore, we can see that the number of rounds during big batches is $O(S_\tau(n) + n\tau/P + w(n)/P)$. Since each round has at most $4P$ steal attempts, we get the desired bound. □

Free steal attempts. Here, each "round" consists of $4P$ consecutive free steal attempts (during which no big batch is active). Recall that when a worker becomes trapped, it places $\Theta(\tau)$ dummy nodes on the bottom of its core deque. We say that a round is **bad** if, at the beginning of the round, some trapped worker's core deque is empty (does not have any core nodes or dummy nodes). Otherwise, a round is **good**. Note that bad rounds only occur while

some batch is executing; otherwise no worker is trapped. We bound good and bad rounds separately.

Good rounds do not have the problem of too much potential being concentrated in a suspended data structure node of a trapped worker. During a good round, there is more potential in the dummy nodes than the suspended data structure node itself, and steal attempts reduce potential.

Lemma 3.17. *The number of good rounds is $O(\mathcal{S}_G)$ in expectation and $O(\mathcal{S}_G + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$. Therefore, the number of free steal attempts in good rounds is $O(P\mathcal{S}_G)$ in expectation and $O(P\mathcal{S}_G + P\lg(1/\epsilon))$ with probability at least $1 - \epsilon$.*

Proof. During a good round, there are $4P$ total steal free steal attempts, and thus by the alternating steal policy, half of these ($2P$) are from core deques. Since no trapped worker's deque is empty when the round begins, we can apply lemma 3.12 to conclude that each round decreases the core potential by a constant factor with a constant probability; being interrupted by a big batch only decreases the potential further. We can then apply lemma 3.13 to conclude that there are $O(\mathcal{S}_G)$ rounds show the requisite bound; multiplying by P gives the bound on the number of free steal attempts during good rounds. □

We can now bound the number of bad rounds using the following intuition. The number of bad rounds is small since small batches have small spans, chances are most small batches finish before any trapped worker runs out of dummy nodes.

Lemma 3.18. *The total number of free steal attempts during bad rounds is $O(n\tau)$ in expectation.*

Proof. A worker p places $\Theta(\tau) = b\tau$ dummy nodes, for constant b, on its core deque when it becomes trapped. There is a bad round if its core deque is stolen from at least $b\tau$ times before p becomes free again. There are two cases:

Case 1: worker p is trapped for $k\tau$ rounds, for some constant k; applying a Chernoff bound, during $k\tau$ rounds, each core deque is stolen from $< k_1\tau + k_2 \lg P$ times with probability $> (1 - 1/P^2)$ for appropriate settings of constants k_1 and k_2. If $\tau \geq \lg P$ and $b = k_1 + k_2$, then p's deque runs out of dummy nodes with probability $< 1/P^2$. Since there can be at most $k\tau$ bad rounds, we get the expected number of bad rounds $O(\tau/P)$.

Case 2: worker p is trapped for more than $k\tau$ rounds, for constant k. From lemma 3.6, we know that p is trapped for at most 2 batches, say A_1 and A_2. Therefore, at least one of A_1 and A_2, say A_i, must be active for more than $k\tau/2$ rounds. We first bound the number of rounds during which A_i can be active, with high probability. If A_i is active throughout a round, then there are at least $2P$ steal attempts from batch deques during the round r (since half the free steal attempts hit batch deques) and lemma 3.15 applies. If a batch starts or ends during r, its potential decreases by a constant factor trivially. We can then apply lemma 3.13 to show that with probability at least $1-\epsilon$ the batch A_i is active for $O(s_{A_i} + \lg(1/\epsilon)) = O(\tau + \lg(1/\epsilon))$ rounds, since A_i is not long. (p is waiting for the small batch A_2; therefore, the preceding batch A_1 is also not long.) We know that $O(\tau + \lg 1/\epsilon) < k_1\tau + k_2 \lg 1/\epsilon$ for some constants k_1 and k_2; we set $\epsilon = 1/P^2$ and $k/2 = k_1 + 2k_2$. The probability that A_i is active for $k\tau/2$ rounds is at most $1/P^2$. There can be at most $P\tau$ bad rounds for A_i, since each round takes at least one time step, and a small batch has at most $P\tau$ work. Therefore, the expected number is at most $O(\tau/P)$.

Adding over the n batches that can trap a worker, and over P workers, gives us $O(n\tau)$ in total. □

Corollary 3.19. *Ignoring the batch-setup overhead, the expected number of steps taken by free processors when no big batch is active is* $O(T_1 + w(n) + n\tau + P\mathcal{S}_G)$.

Proof. A free worker is either working (at most $T_1 + w(n)$ steps) or stealing (bounded by lemmas 3.17 and 3.18). □

Trapped steal attempts and batch-setup overhead. We next analyze the steal attempts by trapped workers during small batches. The key idea is as follows. Recall that a worker is trapped by a batch A only if it has a pending data structure node whose operation record is being processed by A or will be processed by the succeeding batch A' (see lemma 3.6). If more than $P/2$ workers are trapped on a A, then either A or A' must be popular, in which case A is called big. Therefore, at most $P/2$ workers a be trapped by a small batch.

Lemma 3.20. *The expected number of processor steps taken due to batch-setup overhead and trapped steal attempts is* $O(T_1 + w(n) + n\tau + P\mathcal{S}_G + PS_\tau(n))$.

Proof. The batch-setup overhead is $O(P)$ per batch. After it launches, each batch executes for at least 1 time step and only one batch executes at a time. For big batches, during this one time step, the workers perform P steps of either work (bounded by $T_1 + w(n)$) or big batch steals (bounded by lemma 3.16). We can amortize the batch-setup overhead against these P steps. For small batches, at least $P/2$ processors are free and again they perform either work or free steals (bounded by corollary 3.19), and we can amortize the batch-setup overhead against this quantity. Adding these gives us the bound on batch-setup overhead.

Even if we pessimistically assume that trapped workers do nothing but steal during small batches, since at least half the workers are free, we can amortize these steals against the steps taken by free workers which either work or steal or perform batch setup steps. □

Overall running time. We can now bound the overall running time. We combine the bounds from lemmas 3.16 to 3.18 and substitute $\mathcal{S}_G = T_\infty + m\tau$ and divide by P (since there are P workers performing these steps) to prove theorem 3.8.

Proof of Theorem 3.8. From lemmas 3.16 to 3.18 and 3.20, we know that the expected number of big-batch steal attempts is $O(n\tau + PS_\tau(n) + w(n))$, free steal attempts is $O(PT_\infty + Pm\tau + n\tau)$, and trapped steal attempts is $O(T_1 + w(n) + n\tau + P\mathcal{S}_G + PS_\tau(n))$. The total batch-setup overhead is $O(T_1 + w(n) + n\tau + P\mathcal{S}_G + PS_\tau(n))$. Adding the total work and dividing by P gives the result. □

We can now set an appropriate value for τ to get the bound on Batcher performance. This corollary is equivalent to theorem 3.1.

Proof of Corollary 3.9. We get this bound by setting τ to be equal to the data structure span $s(n)$. Recall that long batches are defined as batches with batch span longer than τ, and τ-trimmed span $S_\tau(n)$ is defined as the sum of the spans of all long batches. Recall, also, from the definition of the data structure span $s(n)$ is defined as follows: For any sequence of batches comprising a total of n data structure nodes, such that no batch contains more than P data structure nodes, $s(n)$ is the worst case span of any batch individual A that also has parallelism limited by $w_A/s_A = O(P)$.

Since the program has a total of n data structure nodes, and Batcher only generates batches with at most P data structure nodes, the only batches with $s_A > s(n)$ are those where $w_A/s_A = \Omega(P)$. Now, say L is the set of long batches. For all $A \in L$, we have $w_A = \Omega(Ps_A)$, since all other batches have span smaller than $s(n)$, hence also smaller than τ, since $s(n) = \tau$. That is, the long batches are all batches with large parallelism. Therefore, $w(n) \geq \sum_{A \in L} w_A = \sum_{A \in L} \Omega(Ps_A)$. Since $S_\tau(n) = \sum_{A \in L} s_A$, we conclude that $w(n) = \Omega(PS_\tau(n))$, or $w(n)/P = \Omega(S_\tau(n))$. The bound follows from theorem 3.8 as $w(n)/P$ dominates $S_\tau(n)$. □

3.4 Empirical Evaluation

Our prototype implementation was built by modifying the Cilk-5 [77] runtime system. The main difference between the theoretical and the practical design is within the LAUNCH-BATCH operation (figure 3.4). Because we are running on a relatively small number of cores (compared to the size of the data structure), we used a sequential implementation for the status changes status changes (lines 1 and 4) and the compaction (line 2).

Our preliminary evaluation uses a skip-list data structure. Our batch insert (BOP) into the skip list has three steps. (1) build a new skip from a set of records, (2) perform searches for these nodes in the main skip list, and (3) splice the new list into the main list. Since the new list is small (batch size), we perform steps 1 and 3 sequentially, whereas the searches into the large main list in step 2 are performed in parallel. The core program is simply a parallel-for loop that inserts into the skip list in each iteration (e.g., as in figure 3.1). Note that this is a bad case for Batcher since all of the work happens within the data structure, and hence the overheads are tested.

In all our experiments, we first initialize the skip list with an initial size. We then timed the insertion of $100,000$ additional elements into this skip list. To simulate bigger batches without the NUMA effects of going to multiple sockets, each BATCHIFY call creates 100 insertion records. We compared Batcher with a sequential implementation where all $100,000$ elements are inserted sequentially (without concurrency control).

We conducted experiments on a 2-socket machine with 8 cores per socket running Ubuntu 12.04. The processor was Intel Xeon E5-2687W. The machine has 64GB of RAM and 20MB of L3 cache per socket. For our experiments, we pinned the threads to a single socket on this machine.

These experiments are meant to be only a proof-of-concept (though chapter 4 uses a variant of this runtime system for a real-life application, showing its practical value). Nevertheless, the results indicate that implicit batching is a promising approach, at least for reasonable data structures and large-enough batches. For the particular experiment here, Batcher's performance on 1 processor is comparable to that of a sequential skip list, and hence the overhead is not prohibitive. In addition, Batcher provides speedup when running on multiple processors.

Figure 3.5 shows the throughput of Batcher and a sequential skip list with `initialSize` 20,000, 100,000, 1 million, 10 million and 100 million (e.g. `BAT20000` shows Batcher's with initial size 20,000). For initial size $20,000$ and $100,000$, `SEQ` performs better than `BAT` on a single processor. This is because inserts into small skip lists are so cheap that Batcher's overheads begin to dominate. However, even on these small skip lists, Batcher provides speedup, and outperforms the sequential skip list on multiple processors. For larger skip lists, the inserts get expensive enough that they dominate Batcher's overhead, and Batcher performs comparably with the sequential list even on 1 processor.

More interestingly, Batcher's speedup increases as the skip list gets larger. At size 100 million, Batcher provides a speedup of about 3× on 6 processors, and 3.33× on 8 workers. These results indicate that implicit batching is a promising approach, at least for reasonable data structure and large-enough batches.

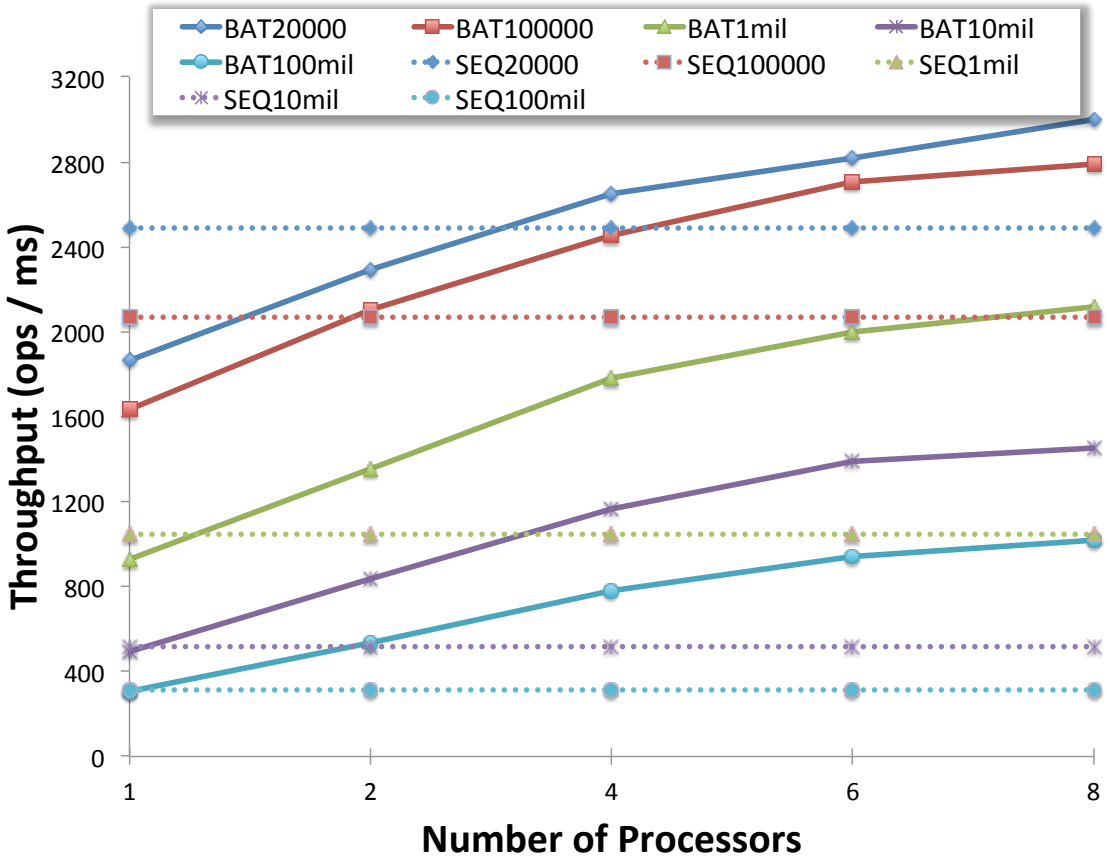


Figure 3.5: Throughput of Batcher and sequential skip list insertion for various initial sizes of skip lists (higher is better).

Flat combining. We view flat combining [88] as a special case of implicit batching where batches execute sequentially. Hender et al. [88] show that flat combining significantly outperforms a good concurrent skip list, at least for certain workloads, providing additional validation for the idea of implicit batching. In our experiments, flat combining and Batcher perform similarly on 1 core. However, the performance of flat combining decreases with

increasing cores (their experiments also show this). In contrast, the prototype Batcher implementation shows speedup.

3.5 Related Work

In addition to flat combining, Batcher most closely resembles various software combining techniques, designed primarily to reduce concurrency overhead in concurrent data structures. In some combining techniques [70, 88, 144], each processor inserts a request in a shared queue and a single processor sequentially executes all outstanding requests later. These works provide empirical efficiency, but we are not aware of any theory bounding the running time of an algorithm using these combiners. Batcher improves upon these techniques by operating on the "request queue" in parallel and by providing runtime theory. Other software-combining techniques include (static) combining trees [83] or (dynamic) combining funnels [167] which apply directly to data structures with composable operations like lock objects, counters, or stacks. These do have a provably $O(\lg P)$ overhead, but do not address more general structures.

Several related mechanisms designed for dynamic multithreading have a grounding in theory. Reducers [79, 123] in Cilk can be used to eliminate contention on some shared global variables, but are not designed to replace a generic concurrent data structure, since they create local views on each processor rather than maintain a single global view. It is also unclear how to analyze reducers that include highly variable amortized costs. Helper locks [3] provide a mechanism that allows blocked workers to help complete the critical section that is blocking them and is not specifically designed for data structures. Conceptually, one can use this mechanism to execute batches; however, directly applying the analysis of [3] leads to worse completion time bounds compared to using Batcher.

Concurrent data structures themselves are widely studied [93]. Most theoretical work on concurrent data structures focuses on correctness and forward-progress guarantees like linearizability [91], lock freedom [89], or wait freedom [90]. While wait-free structures often include a worst-case performance bound, the bound may not be satisfying when applied in the context of an enclosing algorithm. For example, a universal wait-free construction of [48] has a worst-case cost that includes a factor of P, the number of processors, which implies serializing all data structure operations. Experimental studies of various concurrent B-tree data structures alone spans over 30 years of research [16, 18, 80, 84, 106, 107, 108, 109, 111, 112, 115, 150, 161, 160, 162, 171, 34].These results typically fall short of bounds on running time, with [18] being one exception assuming uniformly random accesses.

Several batched search trees exist, including 2-3 trees [148], weight-balanced B-trees [68], and red-black trees [75]. Moreover, some of these data structures [68, 75] exhibit good practical performance.

3.6 Conclusions and Future Work

Batcher brings the use and design of the data structures in parallel programs closer to that of sequential programming. The scheduler provides implicit batching, allowing enclosing programs to essentially treat batched data structures as concurrent data structures. Batcher is theoretically efficient and preliminary experiments indicate it can provide speedup in practice, especially when Batcher's overheads can be amortized by expensive data structure operations.

Several open questions remain in this direction. On the practical side, many more data structures could be written to support Batcher. What data structures are more easily and

efficiently expressed in this manner? Are there real applications that are sped up by use of Batcher? On the one hand, there may be some significant constant factors hidden in the big-O notation. But on the other hand, there are several secondary effects like contention on shared locations (which Batcher largely avoids) that may give it a competitive advantage.

Batcher may be improved by some design variants. It may be possible, for example, to remove or reduce the $O(\lg P)$ overhead with a different communication mechanism. Another design point is when Batcher decides to start a batch. We took the simple approach of starting a batch as soon as any operation is invoked, but there are many possibilities. We might allow threads to notify Batcher of a set of operations and then manually start a batch, which may happen to include operations from other thread as well. Or we could employ a lazy approach — strands that invoke BATCHIFY are simply suspended, perhaps using the mechanism from chapter 5. Batches might only be started after a certain threshold of failed steal attempts, indicating that many strands are suspended waiting on data structure operations. There are many fruitful directions to explore here.

This approach may also be useful to statically threaded programs. A pthreaded program might make calls to Batcher, which operates with a pool of work-stealing threads separate from the pthreads.

Chapter 4

Parallel Race Detection for Fork-Join Programs

As discussed in section 2.5, race conditions are among the most insidious bugs in modern software, since the output is dependent on the order of particular events. Debugging determinacy races[1] is a challenging aspect of parallel programming. To ease the burden, **race detectors** identify instances of determinacy races and the code that causes them.

There are three types of race detection algorithms. In static race detection, the source code of a program is analyzed to identify races. Unfortunately, static race detection is an NP-hard problem [137], and thus such algorithms either resort to identifying false positives or missing some races. A trace-based, or post-mortem, race detection algorithm [138, 46, 5] records the instructions of a computation[2]. After the computation has completed, this trace, which may be exorbitantly large, is analyzed to find races. In "on-the-fly" race detection [132, 71, 154, 155, 19], we analyze a program as it executes. This approach adds overhead to programs, but provides a strong correctness guarantee – both consistency and correctness. In this context this means that for a particular input, these race detectors report a race if and only if the

[1] Hereafter we will simply use the term **race**.

[2] Recall from section 2.1 that a **computation** refers to a program with a specific input

program contains a race on that input. In other words, if different schedules can result in writes being reordered with respect to other accesses to the same memory location.

At every memory access, an on-the-fly race detector retrieves the history of previous accesses to the memory location and asks if they could happen in parallel with the current access. Thus on-the-fly race detectors require two main components. The *memory access history* tracks the previous readers and writers to each memory location. The *SP-Maintenance* algorithm maintains information on the series-parallel relationship between strands in the program – whether or not two strands are logically parallel. Most race detection research has focused on the SP-maintenance component, using very similar access history schemes. Unfortunately, existing SP-maintenance algorithms have performance shortcomings – most execute serially [71], while others have high worst-case overhead [154, 155] or are impractical to implement and limit parallelism [19].

This chapter presents an asymptotically optimal and parallel algorithm for SP-maintenance on fork-join programs, **WSP-Order**. For a given deterministic computation with work T_1 and span T_∞, WSP-Order executes the computation in expected time $O\left(\frac{T_1}{P} + T_\infty\right)$ on P processors. Since executing a parallel program takes $\Omega\left(\frac{T_1}{P} + T_\infty\right)$ without performing SP-maintenance, WSP-Order is asymptotically optimal.

WSP-Order builds directly on the SP-Order algorithm proposed by Bender et al. [19]. SP-Order solves the SP-maintenance problem sequentially with a pair of order-maintenance (OM) data structures, which support constant-time operations. However, it is not immediately clear how to produce an efficient parallel version of SP-Order due to its reliance on the shared OM data structures. Bender et al. [19] propose a significantly more complicated algorithm allowing parallel execution, but it has non-constant overhead.

Rather than modifying SP-Order itself, WSP-Order achieves its efficiency by integrating a modified work-stealing scheduler [29] with modified OM data structures. A key insight is that SP-Order uses OM data structures in a well structured way — concurrent accesses to the data structures do not logically conflict; therefore, in principle, concurrent updates to the data structure can generally proceed without concurrency control. Occasionally, however, the OM data structures undergo **relabel operations**, wherein a large portion of the data structure is modified. These relabels, which are necessary to achieve the optimal performance bounds, do conflict with concurrent operations. Coping with relabels in the OM data structure is the main challenge in parallelizing SP-order and achieving asymptotically optimal performance.

The high-level ideas behind WSP-Order are to (1) forbid concurrent accesses during relabels, but otherwise allow all concurrent OM operations to proceed without any significant concurrency control; (2) use parallelism within relabel operations to ensure that they can finish quickly; (3) modify the scheduler to prioritize relabels; and (4) ensure that relabels do not occur "too frequently" by relabeling more eagerly. As we shall see in later sections, these four modifications are sufficient to achieve provably good performance while still allowing practically efficient implementation. Here:

Combining WSP-Order with an access-history algorithm provides a performant parallel race detection algorithm. Maintaining access history unfortunately requires concurrent updates on reads, so providing guarantees for race detection is difficult without a strong contention model. By assuming a constant-time priority-write primitive, however, CRacer can provide full race detection in expected time $O\left(\frac{T_1}{P} + T_\infty\right)$.

Outline. Section 4.1 describes the SP-maintenance problem in detail and reviews SP-Order, the sequential algorithm upon which WSP-Order builds. We describe the design of

WSP-Order in section 4.2, analyze its performance in section 4.3, and describe empirical results of our implementation in section 4.4. Section 4.5 discusses some related work before concluding in section 4.6

4.1 Series-Parallel Maintenance

This section describes the series-parallel (SP) maintenance problem, sketches how SP-Order solves it, and shows how SP-Order is used to detect races. SP-Order adds no asymptotic work to a computation, but forces a computation to run sequentially.

One way to maintain series-parallel relationship between strands is to build the computation graph during execution. That is, for each newly created strand, we would create a new node and insert the appropriate edges into a graph data structure. While simple, this approach requires exorbitant space and requires a graph traversal at each query.

4.1.1 Graph Labeling

For fork-join computations, various labeling schemes [132, 140] have been devised that avoid maintaining the entire computation graph. These techniques assign two or more labels to each strand at creation time (in Cilk Plus, at a `spawn`) and compare labels to determine series-parallel relationships. In the English-Hebrew labeling [140], for example, two strands x and y are logically parallel if and only if their "English" and "Hebrew" labels *disagree*: $x_e < y_e$ and $x_h > y_h$ or vice versa, where the subscripts denote the labels.

Assigning fixed labels is complicated since the computation graph is expanding dynamically. When choosing the label for the left child of a strand, for example, we do not know how many descendants the right child will have. This problem can be fixed by using large strand

labels [140]. Unfortunately, doing so grows the label size with the number of forks in a program, requiring non-constant time to compare labels.

4.1.2 SP-Order

Keeping fixed labels is actually overkill for this problem – we don't care if labels change, provided the correct ordering is maintained. This is a good fit for **order-maintenance** (OM) data structures, which efficiently maintain a total order of elements. The semantics of a "total order" type can logically be thought of as a linked list, but the implementation need not be a list. These data structures provide an operation `insert(x,y)` to insert a new strand y after an existing strand x, and an operation `order(x,y)`, which returns whether or not x occurs before y in the ordering. SP-Order uses two such structures – one each for the English and Hebrew orderings.

There are several different order-maintenance data structures [58, 57, 20, 183], achieving $O(1)$ amortized[3] cost per operation. Hence SP-Order can maintain SP relationships with only constant-time overhead.

SP-Order [19] uses order-maintenance data structures to maintain two different total orderings related to Nudler and Rudolph's the English and Hebrew labeling scheme [140]. Figure 4.1 shows pseudocode for how SP-Order maintains these orderings for all strands in the computation[4] In the frame of each function F, SP-Order has pointers to elements in English and Hebrew structures representing its currently executing strand. On spawns, it creates nodes for three new strands: the newly spawned function, the continuation strand,

[3]There are worst-case versions of the sequential order-maintenance data structures [57, 20], but those updates are inherently sequential. The amortized versions, however, are much easier to parallelize.

[4]The original description of SP-Order is with respect to a series-parallel parse tree; we adopt an operational description here.

and the strand immediately after the corresponding `spawn` (this last one need only be created for the first spawn of the sync block). These three new elements are inserted after the current strand in both orderings. In the English ordering their order is spawn then continuation then sync. In the Hebrew ordering, their order is continuation then spawn then sync. These orderings are sufficient to determine SP relationships [140]; a strand x logically precedes strand y if and only if x precedes y in both orderings.

On F spawning G :

```
 6  if first − spawn = true
 7      first − spawn = false
 8      create OM-Element for F.sync.e and F.sync.h
 9      OM-Insert(Eng, F.curr.e, F.sync.e)
10      OM-Insert(Heb, F.curr.h, F.sync.h)
11  create OM-Element for G.curr.e, G.curr.h,
    F.cont.e, and F.cont.h
12  OM-Insert(Eng, F.curr.e, G.curr.e, F.cont.e)
13  OM-Insert(Heb, F.curr.h, F.cont.h, G.curr.h)
14  F.curr.e = F.cont.e;
15  F.curr.h = F.cont.h
```

On F passing sync :

```
16  first − spawn = true
17  F.curr.e = F.sync.e
18  F.curr.h = F.sync.h
```

On F calling child H :

```
19  H.curr.e = F.curr.e
20  H.curr.h = F.curr.h
```

On a called child H returning to F :

```
21  F.curr.e = H.curr.e
22  F.curr.h = H.curr.h
```

Figure 4.1: The SP-Order algorithm. SP-Order maintains two OM data structures, *Eng* and *Heb*. For a function F, elements representing the currently executing strand are in $F.curr.e$ and $F.curr.h$. $F.cont.e$ and $F.cont.h$ represent F's continuation strand after the spawning of G. $F.sync.e$ and $F.sync.h$ represent the strand after the corresponding `sync` in F.

We use the SP-Order's SP-maintenance algorithm without modification; therefore, the correctness guarantee follows from correctness of SP-Order (presented in [19]). In addition, while SP-Order was described as sequential algorithm, it works in parallel out of the box if we can insert and query into the English and Hebrew structures concurrently. (The question is performance.)

4.1.3 OM Implementations

To understand WSP-Order we'll need some details about the order-maintenance data structure. As mentioned above, OM data structures can provide both inserts and queries in constant time. To provide this, the structure is divided into a top and bottom level. The bottom-level structures, called **groups**, contain the actual elements of the order (in this case, strands), while pointers to bottom-level structures are maintained in the top-level structure.

Both elements and groups contain (integer) labels. Group labels are unique among all groups, and element labels are unique within a group. The group levels serve to order elements in two different groups; we can think of each element as having two labels L_B and L_T, for bottom-level and top-level, respectively. So queries can be performed in constant time by comparing the labels of two elements and of their respective groups – x precedes y in the total order if and only if (1) $L_T(\mathit{group}(x)) < L_T(\mathit{group}(y))$, or (2) $\mathit{group}(x) = \mathit{group}(y)$ and $L_B(x) < L_B(y)$.

To insert element y after element x, we try to insert y directly after x in x's group, assigning it a label between x and x's successor. If such a label assignment is possible, we are done. In some cases, however, x's label one less than its successor. In this case the group is **full** and requires a **relabel** operation. There are two steps involved in relabel operations: (1) splitting x's group into smaller groups, spacing out the labels; (2) pointers to the new groups are

inserted into the top-level structure, possibly performing operations to ensure the top-level structure is balanced. Note that although labels may change, the ordering of two elements in the structure must always be consistent.

If we make sure that each group contains $\Omega(\lg n)$ elements, the total number of splits and top-level insertions is $O(n/\lg n)$. We can ensure the cost of top-level insertions is $O(\lg n)$. Thus the cost for insertion is $O(1)$ amortized.

4.2 WSP-Order

We now present the WSP-Order algorithm for maintaining SP relationships in parallel. This algorithm originally appeared in [185]. WSP-Order modifies the work-stealing runtime system and OM data structures to allow SP-order to run in parallel. The theoretically-optimal algorithm is presented here; our implementation involves minor changes which we describe in section 4.4.

4.2.1 Overview

Although SP-Order is not inherently sequential, concurrent updates to OM data structures are problematic.

One possible way to parallelize SP-Order would be to use the Batcher scheduler from chapter 3 along with a batched OM data structure. Unfortunately, since most OM operations are constant time, Batcher itself adds $\Omega(\lg P)$ work/time to each data-structure access[5]. Therefore, even with an efficient OM data structure, it seems impossible to achieve an optimal SP-maintenance algorithm using Batcher directly.

[5]Since this $\lg P$ overhead is additive, Batcher's performance theorem is great for high-cost data structures like concurrent search trees. But it falls short on constant-cost data structures like order-maintenance

WSP-Order does better by leveraging the insight that both the OM data structure and SP-Order algorithm are well structured; most concurrent accesses do not logically conflict. When the occasional relabel is necessary, however, a large portion of the data structure is modified and may conflict with concurrent operations.

There are four main design principles for WSP-Order:

1. If no relabel is in progress, all operations can be allowed to proceed without any synchronization. Otherwise some concurrency control is necessary to ensure that concurrent operations do not conflict.

2. Relabel operations can be quite long, blocking inserts and queries. We need to use parallelism within relabels so that they finish quickly.

3. Even with parallel relabels, core work may be blocked during a relabel. Workers may spend too long stealing core work, only to quickly realize they cannot continue until the relabel operation has finishes. Thus we provide runtime scheduler support to guide workers towards helping active relabel operations, prioritizing relabels.

4. Even with parallel relabel operations, relabels may occur "too frequently". We reduce the number of relabels by relabeling more eagerly.

The remainder of this section explains each of these in detail. We first explain the mechanism for reducing concurrency control and the scheduler modifications. Then we describe the implementation of the OM data structure used by WSP-Order. Finally, we present how to perform fewer relabels and how to do relabels in parallel.

4.2.2 Concurrency Control

The main insight of our approach is to utilize the structure of fork/join parallelism to avoid synchronization for most OM operations. Most inserts and queries take a *fast path* which requires very little synchronization. Occasionally (for relabels), operations must use a *slow path*, for which we use a simple locking scheme.

First we note that all insertions in SP-Order insert directly after the strand currently being executed by that worker. Any concurrent insertions involve different strands and thus have no logical conflicts. Further, each insert of y after x need only touch the x's node, i.e. changes are local. As long as no relabel operation is in progress, this means that concurrent inserts can safely operate in parallel. Finally, queries are read-only operations – they only compare labels. Consequently, barring a change to labels or groups, queries and inserts can also occur concurrently.

Thus the main issue is to avoid a relabel overlapping with an insert or query. To do so we use a two-level locking scheme. Each worker has a **local lock** that it must acquire before performing any insert or query. There is also a single **global lock** to protect relabel operations. When a relabel is in progress, the relabel process acquires the global lock as well as *all* the local locks. Therefore, a relabel cannot be concurrent with any insert or query; however, inserts and queries from different workers can proceed concurrently with each other.

Figure 4.2 shows the pseudocode for the underlying INSERT operation used by OM-INSERT. When a worker w is trying to insert element y after element x, it first checks to see if the group that x belongs to is already full (line 4). If so, w either triggers a relabel process (line 7) if one is not in progress (the global lock is free), or it joins the ongoing one (line 9). If the group is not full, w proceeds to acquire its local lock (line 10) and perform the insert

(line 11). If the local lock is not free, a relabel is in progress, and w joins the relabel (line 17). If the local lock is free, the insert always succeeds since the group was not full before the insert. The insert may cause the group to become heavy or full, however, in which case the group is marked as heavy (line 14) and will be split in the next relabel. (We will describe the condition a group being heavy and the relabel process in later subsections.)

There are a few subtleties in the code shown. First, checking if the group is full must be in the loop, since even though w may join an ongoing relabel process, that relabel process may not include g (for instance, the relabel process started before g became heavy). In this case, g remains full after this particular relabel operation finishes, and w will need to recheck g for fullness and possibly start the another relabel operation. Second, the local-lock acquire must also be in the loop, since w may fail to acquire the lock, join the relabel, and must retry the insert again when the relabel ends. Finally, since RELABEL is a parallel operation, the worker who initiated RELABEL's may not be the one who executes its final instruction. Note that it is important to release the global lock at the end of RELABEL (as opposed to in INSERT after START_RELABEL returns), so the global lock must be thread-oblivious, i.e. allowing acquisition and release by different workers.

4.2.3 Prioritizing Relabels with Scheduler Support

Unfortunately, it is not enough to simply parallelize the relabel operation. Since an ongoing relabel prevents other concurrent OM operations, we want an ongoing relabel to finish quickly; moreover, workers blocked on an insert or query should help with the relabel instead of being idle. Prioritizing relabels in this way requires scheduler support much like that in chapter 3. Each worker has a **DS deque** in addition to its **core deque**. Only work associated with relabel operations is contained on the DS deque, while the core deque holds all other

work. When workers steal work from some other worker's core deque, they place it on their own core deque, and when they steal work from some worker's DS deque, they place it on their own DS deque. This scheme differs from Batcher in how we switch to the DS deque and in that there is no "collection" of batch operation records, facilitating a simpler and more concise analysis.

In general, workers prioritize the DS deque, enabling relabels to finishes quickly. In particular, when a worker w starts a relabel (START_RELABEL in line 7), w suspends the current strand, switches to the DS deque to work on relabel, and does not return until discovering that the relabel has finished. Likewise, when a worker w fails to acquire a global or local lock (i.e., a relabel is in progress), w invokes JOIN_RELABEL (in lines 9 and 17), which suspends the current strand, switches to work-steal on the DS deque until the relabel finishes. Finally, when the current deque is empty, w checks the global lock before performing a steal attempt. If the global lock is held, it starts work stealing on the DS deques. If the global lock is free, it operates on the core deque, resuming any suspended strand. If the core deque is empty, it starts work stealing from core deque.

4.2.4 OM Data Structure Implementation

Before we describe the parallel relabel, we must first describe in more detail the implementation of the OM data structure used in WSP-Order.

Bottom-level structure: In sequential versions of OM data structure, the bottom-level structures are typically implemented as linked lists. WSP-Order, however, must iterate over all elements in parallel during relabels. Therefore, in WSP-Order, these are implemented as unbalanced binary tree where each internal node has exactly two children. All the OM

elements are stored in the leaves. Each element has an integer label corresponding to its root-to-leaf path (as in a trie), i.e., 0 for left child and 1 for right child, with the root starting from the high-order bit of a b-bit machine word. Any unused trailing bits are implicitly 0s. Each element also stores its depth in the tree (i.e., the logical label length) and its group identifier.

To insert a new element y after an existing element x, first assign y the same group as x. Next, create a new internal node with x and y as the left and right children, respectively, and splice that node in place of x in the tree. Next, assign y the label induced by its root-to-leaf path by appending a 1 to x's label, i.e., $L_B(y) = L_B(x) + 2^{b-depth(x)}$. Finally, record y's depth and implicitly refine the precision of x's label by incrementing x's depth. It is not hard to see that y's label correctly matches the root-to-leaf path to y and that these updates can be performed in constant time. Note that the label induced by x's path does not change since 0s are appended regardless, so concurrent queries on x are not affected by the update.

The structure is considered **full** when the depth of any node reaches b — in this case, we no longer have space to insert another node immediately after the currently inserted node while keeping the number of bits in the label at most b. At this point, a relabel is triggered, which will change the labels of all elements in this group.

Top-level structure: WSP-Order needs to operate on the top-level structure in parallel, so a structure like Dietz's original tree-based version [58] is most appropriate. Here, we describe the basic sequential structure for concreteness, and while some of the details differ slightly from previously published structures, we are not claiming any new ideas here.

The top-level structure is a balanced binary tree where each internal node has exactly two children. Like the bottom-level structures, all elements are stored in the leaves, and each

element is explicitly labeled according to its root-to-leaf path. The elements (leaves) of the top-level structure are (pointers to) the bottom-level groups. Since the top-level structure can be much larger than bottom-level structures, we must maintain a balanced tree to ensure that labels fit in a machine word. Therefore, the easy insertion algorithm described earlier is not sufficient. On the other hand, all inserts into the top-level structure occur inside relabel operations, so they are protected by the global lock.

Since the labels of elements change whenever its root-to-leaf path changes, rotation-based balancing schemes do not readily apply. Instead, most implementations of the top-level structure use some form of weight balance. For each node, WSP-Order maintains a **size** corresponding to the number of leaves in the node's subtree, and a **depth** corresponding to the distance from the root to the node. We say that a node r is **out of balance** if some specific conditions on $\mathit{size}(r)$ are not met. When a node becomes out of balance, the entire subtree rooted at r must be rebuilt. It is not important to understand the specific condition used — there are many balance conditions that would work, e.g., scapegoat trees [7, 81]. Rebuilding the appropriate subtree can be easily implemented as parallel tree walks, with $O(\mathit{size}(r))$ work, which is sufficient to achieve $O(\log n)$ work per top-level insert [7, 81].

4.2.5 Reducing Relabel Frequency

Even with minimal concurrency control and runtime scheduler support, simply parallelizing the relabel operation (described next[6]) would not be enough. The problem is that there may be as many as $\Theta(n/\lg n)$ relabel operations over the course of n inserts into an OM data structure. It turns out that a parallel relabel operation incurs at least $\Theta(\lg n)$ span[7] and each

[6]We describe reducing the relabel frequency first since it complicates the actual relabel details.
[7]It requires a rebalance of a tree with n elements.

relabel occurs one after another (since only one can occur at a time), all relabels together would incur $O(n)$ span, which can be as bad as $O(T_1)$.

We can reduce the frequency of relabels by being proactive in our relabels. In a typical OM data structure, a relabel splits on the group that is full, inserting these new groups into the top-level structure. Our proactive relabel will split any groups that are "close enough" to full at each relabel. Given a label size of b bits, we say a group is **heavy** if there exist two elements only $b/2$ bits apart (or less). During a relabel we can split all such groups in parallel, avoiding any asymptotic increase to the work or span.

We make two changes to keep track of heavy lists. First, each bottom-level structure has a bit flag to indicate whether the structure is heavy. Second, each worker has a local array that stores a list of heavy groups to be read when performing a relabel. When a worker w performs an insertion that results in a $\geq b/2$-bit label, w performs a test-and-test-and-set on the heavy bit. If w manages to set the bit, then a pointer to this group is added to w's local array of heavy groups; otherwise, w does not retry. This procedure ensures that a heavy group is stored on only one worker's local array and trivially takes $O(1)$ time.

After a relabel, no heavy groups remain, meaning that there is at least $b/2$ space between each element in a group. Each insertion `insert(x,y)` halves the space between x and the immediately following element. Since each worker inserts after its own current strand, we can only halve any given interval once per time step. The result is that after a relabel we are guaranteed $O(\lg n)$ timesteps until another relabel is necessary.

4.2.6 Parallel Relabels

We now describe how to parallelize the relabel operation. Recall that during a relabel, all the heavy groups are **split** — that is, all the elements from a full group are partitioned into new groups. These newly created groups are then inserted into the top-level structure, which must be rebalanced.

Figure 4.2 also shows the high-level control flow for the RELABEL procedure. It first acquires all local locks in parallel. These lock acquisitions are blocking, retrying until successful. Second, all heavy lists from P workers' local arrays are concatenated in parallel using prefix sums (line 3). Next, in parallel, we split all heavy groups. Since groups may be large, each split is further parallelized using two parallel, recursive tree walks. The first walk calculates the size of the subtree of each internal node. (These calculations are performed in postorder, i.e., when returning from recursive tree walks.) Given a size-m bottom-level tree, the second walk splits the tree into $\Theta(m/b)$ shallow, fully balanced bottom-level trees of size $\Theta(b)$ and depth $\Theta(\lg b)$. One method is to write all elements into an array in order (in parallel) using size information at internal nodes to determine array offsets, split the array into subarrays of $\Theta(b)$ elements, and then recursively build shallow balanced trees of these segments. Pointers to these new groups are stored in an array attached to the original group's node, for easy parallel access.

Rebalancing the top-level tree in parallel is more complicated. Since we split the heavy groups into multiple groups, some leaves in the top level tree (corresponding to groups that were just split) now represent multiple groups and the ancestors of these nodes have a different size now. We must be careful when updating sizes to avoid race conditions on the internal nodes. The process is as follows:

1. For each split group in parallel (i.e., a parallel loop over the list of heavy groups), follow the leaf-to-root path up the tree, marking all ancestor nodes via test-and-set (no locks needed because the races are benign).

2. Recursively walk down the tree in parallel, only along the marked paths, updating the sizes of internal nodes in postorder as before. For the base case, the size of each modified leaf is the number of new bottom-level structures into which that group was split.

3. Recursively walk down the tree again in parallel, stopping the recursion at each branch that finds an out-of-balance node. Note that there may be many out of balance nodes, but starting from the root finds all the highest ones.

4. Rebuild all out-of-balance subtrees in parallel. Similar to bottom-level tree splits, we can first recursively build an array of leaves, and then use this array to construct a new balanced subtree.

Finally, all marked nodes are unmarked in parallel, all heavy arrays emptied in parallel, and then all local locks released in parallel before releasing the global lock.

4.3 Theoretical Performance Analysis

This section analyzes the performance of WSP-Order. That is, we consider the total execution time of a computation (with work T_1 and span T_∞) augmented to perform SP-maintenance using WSP-Order. At a high level our analyzes analyzes two distinct "phases" of computation. We separately analyze time spent in these phases. During **core phases** only core work (the original computation) and fast inserts and queries occur. When a relabel begins[8], the

[8]We say a relabel begins when the global lock is acquired.

current core phase ends and a **relabel phase** beings. The next core phase beings when this relabel finishes.

As long as inserts and queries (apart from rebalances) take $O(1)$ time, it is not hard to see that the total time spent in core phases is $T_P = O(T_1/P + T_\infty)$ in expectation for a program with T_1 work and T_∞ span on P processors, following from a standard work stealing analysis [29]. The challenge is stitching together the core phases and the relabel phases. The crux of the argument is to leverage the fact that rebalances are provably infrequent, and hence the number of relabel phases is $O(T_P/\log n)$. We then apply a work-stealing analysis to each relabel phase to conclude that they also take $O(T_1/P + T_\infty)$ time overall.

4.3.1 Performance Model

For our theoretical analysis, we make the following modeling assumptions. These assumptions are common in the related literature but not always called out explicitly.

First, our base model is a transdichotomous RAM [74], meaning that (1) b-bit machine words support standard arithmetic and bit operations in constant time, and (2) all pointers fit in a single word. Therefore $b > \lg n$, and all top-level labels fit in a constant number of machine words. Similarly, $b > \lg P$. Second, we assume that all processors operate at the same speed, e.g., that workers are not swapped out to run an operating-system task. Note that this assumption is for performance only, not correctness. Third, for SP maintenance and work stealing, the basic atomic primitive is a compare-and-swap (CAS). We assume that each CAS completes (possibly failing) in constant time regardless of contention. Finally, for our lock objects, which can be implemented using CAS, we assume further that there is some degree of fairness. In particular, if worker w_1 is spinning on a lock, and worker w_2 releases the lock,

then we assume that w_1 acquires the lock before w_2 can reacquire it. In WSP-Order, there is never more than one worker spinning on a lock, so this assumption is reasonable.

4.3.2 Core Phases

We argue that OM operations during core phases are fast and bound the total time in core phases.

Lemma 4.1. *Each data-structure operation performed during a core phase completes in $O(1)$ time and does not conflict with any other operations.*

Proof. By definition, during the core phase the global lock is not held, and no rebalances occur. Thus, an insert operation consists of 1) acquiring the processor's local lock, 2) performing an insert into a bottom-level structure, 3) potentially performing a test-and-set on a group and adding a heavy group to the processor local array of heavy groups, and 4) releasing the local lock. Since there is no competition on the locks when the global lock is not held and test-and-sets are constant-time, each step takes $O(1)$ time. All inserts add new elements either after the element representing currently executing strand in the OM structures. Since each strand is executed by a different processor, concurrent inserts operate with respect to different insertion points and there are no conflicts during inserts. Queries are similar, modulo the details of the query itself which also takes $O(1)$ time. To see that inserts do not conflict with queries, it is enough to observe that 1) all queries by SP-order query elements that have already been inserted into the OM structures, and 2) no insert changes the label of any existing elements, unless it triggers a rebalance. □

Lemma 4.2. *Consider a P-processor execution of a series-parallel computation augmented to perform SP-maintenance. Let T_1 and T_∞, respectively, denote the work and span of the*

underlying a posteriori *computation DAG, i.e., not counting the SP-maintenance operations. Then the total time spent in core phases, including SP-maintenance operations, is* $T_P = O(T_1/P + T_\infty)$ *in expectation.*[9]

Proof. First, augment every node in the computation DAG by $O(1)$ to reflect the worst-case time of any OM operation apart from relabels (with constants chosen according to lemma 4.1). We refer to this DAG as the augmented core DAG.

Observe that, according to a standard work-stealing analysis [29], the augmented core DAG executes in $O(T_1/P+T_\infty)$ expected time on P processors. It remains only to confirm that the execution of core phases is no worse than executing the augmented core DAG. In particular, consider the execution with all relabel phases elided. By lemma 4.1, all data-structure operations during this subexecution take $O(1)$ time. Moreover, when a processor starts working on its DS deque (only during a relabel phase), it simply suspends works on its core deque and resumes where it left off at the start of the next core phase. The execution thus corresponds to a work-stealing execution of the augmented core DAG, except that some nodes (finished during relabel phases) may be removed for free. Specifically some processors (those neither stealing nor performing data-structure operations) may continue to make progress on their core deques during relabel phases, which only improves the time bound of core phases. □

4.3.3 Relabel Phases

We next analyze the work and span of relabel phases, which we can use to get total time spent in relabel phases by applying a work-stealing analysis. We will subdivide the phases further to consider the running time.

[9] In fact, this bound can be extended to a high probability bound as in [29]. Specifically, with probability at least $1 - \epsilon$, the time is $O(T_1/P + T_\infty + \lg P + \lg(1/\epsilon))$.

Lemma 4.3. *The amortized work complexity of each order-maintenance insert is* $O(1)$.

Proof. This analysis is very similar to the standard order-maintenance analyses, e.g., [57]. The only difference is in our parallel implementations of the structures.

Consider any bottom-level tree that is split, i.e., that is heavy at the start of the relabel phase, and m be the number of elements in the tree. The split is implemented as several (parallel) tree walks, so the work is asymptotically the number of nodes in the tree. By construction, each internal node has 2 children, so the number of internal nodes is less than m. The total work is thus $O(m)$.

To get $O(1)$ work per element, we need to charge this $O(m)$ work against $\Omega(m)$ new insertions. Again consider the bottom-level tree. When this tree was created (the result of a previous split), it had $m' = \Theta(b)$ elements[10], and it was as balanced as possible, i.e., having depth $\lg b + O(1)$. Since it is now heavy, its depth has grown to $b/2$, meaning that it has experienced at least $\Omega(b)$ inserts. We charge the cost of the split against these insertions. That is, we divide the $O(m)$ cost by the $m - m'$ new insertions. Since $m' = O(b)$ and $m - m' = \Omega(b)$, we conclude that $m - m' = \Omega(m)$. The work per insertion is thus $O(m/(m - m')) = O(1)$.

The analysis of rebalancing the top-level structure is similar to that of scapegoat trees [81], where the amortized work of an insert is $O(\lg n)$ as long as the cost of rebuilding a subtree is linear in its size. In WSP-Order, the parallel rebalance process has two components (1) following leaf-to-root paths for each top-level insert, giving a worst-case cost of $O(\lg n)$ per top-level insert, and (2) tree walks, giving cost linear in size of the subtrees walked, which yields $O(\lg n)$ amortized work per the standard analysis. Adding these gives $O(\lg n)$ amortized work per top-level insert. Since each bottom-level structure contains $\Omega(b)$ elements,

[10]Recall that b, the number of bits in an integer, satisfies $b = \Omega(\lg n)$.

the number of top-level inserts is $O(n/b)$ and giving the cost per insertion of $O((1/b)\lg n) = O((1/\lg n)\lg n) = O(1)$. □

Lemma 4.4. *The span of each rebalance is $O(b)$, where b is the number of bits in a machine word.*

Proof. The algorithm first acquires all P local locks. Since local locks are fair, it never has to wait for more than 1 operation to finish before it gets a lock and all non-rebalance operations finish in $O(1)$ time. Therefore, it can acquire all local locks in parallel with $O(\lg P)$ span. Concatenating each worker's local list of heavy nodes and then splitting these list in parallel can be done with span $O(\lg P + \lg n)$. Next, bottom-level splits take $O(b)$ to perform divide and conquer on the trees. Finally, the top-level rebuilding has $O(\lg n)$ span to walk up the tree and $O(\lg n)$ span to perform the rebuilding tree walks. Adding these up gives $O(\lg P + \lg n + b) = O(b)$ by the transdichotomous assumption. □

To bound the time in relabel phases, we subdivide the phases further. A **busy subphase** occurs while at least $P/2$ processors are still working on the core deques, i.e., since the beginning of this relabel phase, they have neither stolen nor tried to acquire a the local lock to perform data-structure operations. The remainder of the relabeling phase is the **stealing subphase**, i.e., when most processors are either working off or trying to steal from DS deques. Note that each relabel phase has at most one busy subphase and one stealing subphase — the busy subphase starts at the beginning of the relabel phase, and the stealing subphase persists to the end. But it is possible for a relabel phase to be entirely a busy subphase or entirely a stealing subphase.

Lemma 4.5. *Consider a relabel phase consisting of a w-work rebalance. The expected time spent in the stealing subphase is $O(w/P + b)$, where b is the size of a machine word.*

Proof. By definition of a stealing phase, at least $P/2$ processors are performing work stealing on the DS deque. We can thus apply the work-stealing theorems to a dag with $w + P$ work and b span (lemma 4.4), where the P work comes from the work of acquiring/releasing all P local locks. This gives expected time $O((w + P)/P + b) = O(w/P + b)$. □

4.3.4 Total Time across Relabel Phases

Consider the time bound in lemma 4.5. The work term, $O(w/P)$, is good — the work is low (according to lemma 4.3) and the term implies linear speedup. The problem is the span term of $O(b)$. Specifically, if there are too many relabel phases, e.g., n/b of them, then all we have concluded is that the total time spent in relabel phases is $O(n)$, and we would be better off just running sequentially.

To get a good bound on runtime, we argue next there are not too many relabel phases. We do so by arguing that core phases have to be long.

Lemma 4.6. *Let T_P be the total time spent in core phases. Then there are at most $O(T_P/b)$ relabel phases.*

Proof. In 1 timestep of a core phase, each worker can perform at most one insert into the bottom-level structures. Since each insert is to a different node in the tree, the depth of a bottom-level tree increases by at most 1 per timestep.

In addition, at the start of a core phase, all bottom-level trees have depth at most $b/2$. In particular, any trees deeper than $b/2$ are marked as heavy and split during a relabel phase. When split, any new bottom-level trees are built with a height of $\lg b + O(1) < b/2$ for sufficiently large b. Since, by definition, a core phase ends when some bottom-level tree

reaches depth b, each core phase requires at least $b/2$ time. It follows that the number of core phases (and hence the number of relabel phases) is at most $O(T_P/b)$ □

We are now ready to sum the time across all relabel phases. We consider the busy and stealing subphases separately.

Lemma 4.7. *The total time in busy subphases is $O(T_1/P + T_P/b)$, where T_1 is the work of the computation DAG not counting data-structure operations, P is the number of processors, and T_P is the total time of the core phases.*

Proof. By definition, a busy phase is the time during which at least $P/2$ processors are working on the main computation. Ignoring failed lock acquisitions, each step makes $\Omega(P)$ progress towards the work of the augmented core DAG. Unfortunately, some steps may be unproductive, namely when the local lock acquisition fails. But this can only happen once per processor per relabel phase, after which point the worker shifts to the DS deque. Moreover, all attempts to acquire the lock take $O(1)$ time. The total work, including failed lock acquisitions, is $O(T_1 + PT_P/b)$, where the T_P/b is the number of phases from lemma 4.6. Dividing by the $P/2$ yields the claim. □

Lemma 4.8. *The total expected time in stealing subphases is $O(n/P + T_P)$, where n is the number of strands in the computation DAG, P is the number of processors, and T_P is the total time of the core phases.*

Proof. Applying lemma 4.5 across relabel phases, the total time across all k stealing phases is $O\left(\sum_{i=1}^{k}(w_i/P + b)\right)$, where w_i is the work of the ith stealing phase. By lemma 4.3, $\sum_i w_i =$

$O(n)$. By lemma 4.6, the number of phases is $k = O(T_P/b)$. We're thus left with

$$O\left(\sum_{i=1}^{k}(w_i/P) + \sum_{i=1}^{k} b\right) = O\left(n/P + (T_P/b)b\right)$$

which proves the claim. □

4.3.5 Total Time for SP Maintenance

Finally, we get the overall running time by adding the time for core phases and relabel phases together.

Theorem 4.9. *Consider P-processor execution of a series-parallel computation augmented with WSP-Order. Let T_1 and T_∞, respectively, denote the work and span of the underlying* a posteriori *computation DAG, not counting the SP-maintenance operations. The completion time, including SP-maintenance operations, is $O\left(\frac{T_1}{P} + T_\infty\right)$ in expectation.*

Proof. From lemma 4.2, the total time in core phases is $T_P = O(T_1/P + T_\infty)$ in expectation. Adding the time in busy and relabel subphases as given in lemmas 4.7 and 4.8, we get a total time of $O(T_P + (T_1/P + T_p/b) + (n/P + T_P)) = O(T_1/P + T_\infty)$ since $n = O(T_1)$. □

Since executing the computation without maintaining series-parallel relationships also takes $\Omega(\frac{T_1}{P} + T_\infty)$ time, this bound is asymptotically optimal, in expectation.

4.3.6 Performance of Full Race Detection

As described in the introduction, in addition to SP-maintenance, a parallel race detector must also maintain an access history. Our access history, briefly overviewed here, is similar to that of Mellor-Crummey's [132] race detector. As Mellor-Crummey shows, it is sufficient

for the access history to record the last writer w and two readers — the "left-most" reader lr and the "right-most" reader, with respect to the computation DAG[11]. When a strand s writes to ℓ, the race detector now performs three queries in the SP-maintenance structure, reporting a race if any s is logically parallel with any of lr, rr, or w. If not, s becomes the last writer. If s reads ℓ, s is only compared against w to find a race as before. However, additional queries are necessary to see if s is the new leftmost reader lr or rightmost reader rr. Since each query SP-Order takes $O(1)$ time, this race-detection operation takes $O(1)$ time per memory access.

There is, however, one subtlety. Strands also update the access history if certain conditions are met, recording their identities. Since concurrent writers only occur in the presence of a race, contention on the last writer is not a performance problem. The readers, however, are another story. Multiple strands reading ℓ simultaneously may be "left of" ℓ's current leftmost reader lr. This means that we need some concurrency control on the access history to avoid losing important updates. Note that this is an issue in all race detectors that run in parallel.

Fortunately, the "left of" (and "right of") relations induce total orders, and only the "left-most" ("rightmost") reader's update need be performed. It is possible to implement these relations using a priority-write primitive [170]. Assuming priority updates complete in $O(1)$ time, which seems to match performance when the number of locations being updated is large [170], updating the access history only adds $O(1)$ work per read. Under that assumption, our race detection algorithm, including the access history, runs in $O(T_1/P + T_\infty)$ expected time.

[11] We consider a spawned strand to be the left child, and the continuation to be the right child.

4.4 Empirical Evaluation

This section evaluates the practical performance of CRacer, a parallel race detector using WSP-Order for SP-maintenance. We briefly overview the implementation of CRacer and present experimental results that evaluate CRacer's overhead and scalability. The results show that a program augmented with WSP-Order (but not access history) generally runs less than 2× slower compared with the baseline running without race detection. Full CRacer also scales well, tracking the speedups obtained by the baseline and outperforming **Cilkscreen** [103], a well-engineered, state-of-the-art serial race detector from Intel when running on two or more cores. Finally, we also measured the frequency of relabel operations, and the results indicate that relabels occur infrequently and the eager splitting of heavy groups can indeed help.

4.4.1 Overview of Implementation

Our parallel race detector, CRacer, contains multiple components:

1. the WSP-Order algorithm, including a modified work-stealing runtime system as described in section 4.2.3 as well as an implementation of the OM data structure that does parallel relabels and eagerly splits heavy groups
2. compiler instrumentation for Cilk Plus's parallel constructs and memory accesses
3. an implementation of the access history

The runtime modifications for WSP-Order are based on an implementation of Batcher (see chapter 3), which was originally developed by modifying MIT Cilk [77], and we have ported

it to Cilk Plus by modifying the Cilk Plus runtime. The instrumentation for Cilk Plus's parallel constructs resembles previous work on tool annotations for Cilk Plus [182]; we have used the instrumentation developed for Cilkprof [165], which is implemented in a branch of LLVM/Clang compiler that contains Cilk Plus language extensions [104]. This branch of LLVM/Clang also implements ThreadSanitizer [166], which provides instrumentation for memory accesses. CRacer's access history is maintained with a word (four-byte) granularity. We have done so for practical efficiency reasons, but it means that, CRacer may falsely report a race between strands that access two distinct bytes in the same four-byte granularity. Since most shared variables are at least four bytes (in our experience), we feel this is a worthwhile trade-off.

Our implementation differs from the theoretical design in section 4.2 in a few details. First, in the theoretical algorithm, if the global lock is held, workers always join a relabel operation when they steal. In our implementation, a stealing worker joins a relabel in progress with 50% probability. This scheduling strategy provides the same theoretical guarantees, but the full proof requires an analysis similar that in chapter 3. Second, we have implemented the bottom-level groups in the OM data structure as linked lists rather than trees, and the way we assign labels and marking lists heavy treats the lists as flattened trees. Thus the splits of the bottom-level groups are sequential. Third, instead of performing several traversals of the upper-level tree to determine size information, we walk up the tree only once, using atomic fetch-and-add instructions to increment the size of each node. Fourth, as mentioned in section 4.3.6, parallel readers may update the access history for a given location concurrently, and the theoretical algorithm assumes the use of priority write [170]. In practice, we simply acquire a lock whenever we need to update the location of in the access history, with separate locks for the last writer, left-most reader, and right-most reader per memory location. The

	base	WSP-Order		full(1)		full(16)		Cilksan		Cilkscreen	
matmul	15.69	16.22	(1.03×)	499.61	(31.84×)	31.79	(2.03×)	408.49	(26.04×)	922.23	(58.78×)
cilksort	3.20	3.38	(1.06×)	113.29	(35.40×)	10.81	(3.38×)	73.71	(23.03×)	99.82	(31.19×)
fft	18.34	32.73	(1.78×)	878.17	(47.88×)	81.34	(4.44×)	523.94	(28.57×)	991.29	(54.05×)
heat	7.33	7.73	(1.05×)	1026.70	(140.07×)	79.55	(10.85×)	726.89	(99.17×)	1202.29	(164.02×)
cholesky	5.96	7.44	(1.25×)	687.88	(115.42×)	45.45	(7.63×)	666.66	(111.86×)	962.34	(161.47×)

Table 4.1: Execution times for five benchmarks, in seconds. The `WSP-Order` column shows the execution time while performing WSP-Order but without any memory instrumentation overhead. The `full(16)` column shows execution time while performing full race detection on 16 cores, while all other columns show sequential execution time.

last three modifications potentially reduce the theoretical parallelism of the computation. However, they improve the overall performance in practice due to lower overheads.

Benchmarks. We evaluated CRacer on five benchmarks included with the Cilk-5 distribution [77]. The `matmul` program computes the product of two matrices using a divide-and-conquer algorithm, `cholesky` perform Cholesky decomposition on a matrix, `fft` computes a fast-fourier transform, `heat` performs a heat diffusion simulation, and `cilksort` sorts an array using parallel mergesort.

We ran our experiments on an Intel Xeon E5-2665 with 16 2.40-GHz cores on two sockets; 64 GB of DRAM; two 20 MB L3 caches, each shared among 8 cores; and private L2- and L1-caches of sizes 2 MB and 512 KB, respectively. Running times were averaged over five runs and are presented in seconds. The standard deviation was within 2% of the mean for most configurations, with a maximum of 8% for `cholesky`.

4.4.2 Overhead and Scalability

To evaluate the overhead of CRacer, we compare the following configurations:

- **baseline:** execution without race detection;

- **WSP-Order:** execution augmented with WSP-Order but not the access history management.
- **full:** execution running CRacer with full race detection (i.e., including both WSP-Order and the access history).

We obtain the WSP-Order configuration of CRacer by turning on only the instrumentation for Cilk Plus's parallel control constructs but not memory accesses, allowing us to measure the overhead of only SP-maintenance (including relabel operations), but not include overheads due to access history management.

We also compare with the execution times of these benchmarks running with **Cilksan**, a serial race detector that implements the SP-bags algorithm [71] for SP-maintenance, and with **Cilkscreen** [103], a well-engineered, state-of-the-art serial race detector from Intel that implements the SP-bags algorithm but instrumented using PIN [128], a binary instrumentation framework from Intel, instead of compiler instrumentation. We have implemented Cilksan for fair comparison, since unlike Cilkscreen, Cilksan uses the same compiler instrumentation and access history management as CRacer; the only difference between Cilksan and CRacer is the SP-maintenance algorithm used.

Table 4.1 presents running times of the different configurations. First, note that the overheads due to WSP-Order (SP-maintenance only) shows minimal overhead compared with the baseline. The benchmark with the highest overhead is `fft`, with 1.78 times overhead. This indicates that WSP-Order can be implemented efficiently. Most of the CRacer overhead actually comes from the management of access history — full CRacer incurs another 30–132× overhead, with `heat` being the worst case. This is expected because the WSP-Order performs additional work only at parallel control constructs and function boundaries, whereas

		P				
		2	4	8	12	16
matmul	base	2.00	3.99	7.97	11.88	15.64
	WSP-Order	1.98	3.95	7.87	11.68	15.56
	full	2.00	3.98	7.95	11.82	15.71
cilksort	base	1.99	3.98	7.81	11.17	13.37
	WSP-Order	1.98	3.94	7.71	10.82	13.31
	full	1.92	3.51	6.25	8.19	10.48
fft	base	1.77	3.43	6.12	8.16	9.60
	WSP-Order	1.80	3.31	6.06	8.20	10.00
	full	1.71	3.27	6.43	8.81	10.80
heat	base	1.99	3.63	5.22	7.09	6.71
	WSP-Order	1.98	3.56	5.44	6.62	7.33
	full	2.06	3.86	7.43	10.71	12.91
cholesky	base	1.99	3.96	7.83	11.31	14.69
	WSP-Order	1.86	3.54	6.65	8.73	10.28
	full	1.98	3.87	7.64	11.44	15.13

Table 4.2: Speedup over the sequential version when running with different configurations. For each configuration, the speedup is computed with respect to the running time of the same configuration running on one core.

the access history management incurs additional overhead on every memory access, and for most applications, the number of memory accesses far exceeds the number of `spawn`, `sync`, and function boundaries. The overhead for `heat` is particularly bad, for CRacer as well as Cilksan and Cilkscreen, since compute-to-memory-access ratio is low.

Across all benchmarks, Cilksan outperforms Cilkscreen, so we compare to Cilksan. The ratio of CRacer execution time to Cilksan's is generally less than 1.5. As column `full(16)` shows, however, CRacer utilizes parallelism to significantly outperform Cilksan when running on 16 cores. In fact, as long as the application has some parallelism, CRacer outperforms Cilksan on 2 or more cores.

To evaluate how well WSP-Order and CRacer scale, table 4.2 shows the speedup for each benchmark as we vary the number of cores used. For most benchmarks, our implementation of WSP-Order scales with the baseline, verifying the theory presented in sections 4.2 and 4.3.

	fft				heat				cholesky			
threshold	relabels	median	max	total	relabels	median	max	total	relabels	median	max	total
0.25	1567	18	40	32207	24	3	4	54	22	4	5	66
0.375	1576	15	27	21905	24	1	1	24	22	2	4	56
0.5	1614	19	22	20270	24	1	1	24	22	1	2	31
0.625	1642	12	22	18959	24	1	1	24	22	1	2	31
0.75	1725	7	17	17345	24	1	1	24	22	1	2	24
0.875	2838	4	11	11285	24	1	1	24	22	1	1	22
1	7985	1	3	8090	24	1	1	24	22	1	1	22

Table 4.3: The effects of varying the heavy threshold when running CRacer 16 cores. The first column for a given benchmark shows the number of relabels. The other columns provide information about the number of heavy groups — the median and maximum for individual relabels, and the total number of heavy groups split across all relabels.

The exception is `cholesky`, which executes many spawns but only a small amount of work in each strand. In addition, the CRacer also scales as the number of processors increases, generally at the same rate as the baseline program.

4.4.3 Detailed Breakdown of Overhead

Figure 4.3 shows a breakdown of the overheads due to various components of CRacer for the `fft` benchmark. Due to lack of space, we present this breakdown only for `fft`. We chose `fft` due to the relatively high number of inserts and relabels required, which allows us to show meaningful results here. The x-axis shows the number of cores used, and the y-axis shows the aggregate processing time (i.e., number of cores used $\times$ wall-clock time) in seconds. As the WSP-Order line shows, access history contributes the bulk of the overhead. The other three lines break down the overheads of various components of maintaining the access history. As explained in section 4.3.6, the access history records three strands per memory location and on every access, the race detector queries both OM data structures with the current strands and all three strands, updating the access history as necessary. As shown in breakdown, half of the overhead of access history management is due to the 6 queries and the other half is due to checking and updating the access history. Moreover, whenever a strand s tries to

update the shadow memory, it must acquire a lock. This locking overhead (the gap between lines `nolocks` and `full`) appears to be relatively small.

4.4.4 Effect of Eagerly Splitting Heavy Groups

We now analyze the effect of our modification to OM data structure where instead of only splitting full bottom-level groups, we eagerly split heavy groups. Recall that a tree is full when its depth reaches b and we set the heavy threshold as 50%; that is, a tree is considered heavy when it has depth $b/2$. In practice, we have implemented the bottom-level groups using lists, which means that we mark a list heavy when we discover a label with $b/2$ bits used before trailing 0s. We ask a few different questions here: (1) Does the eager splitting reduce the total number of relabels? (2) How many more splits do we get due to eagerly splitting trees? (3) What effect does the heavy threshold have on the number of relabels and the number of splits?

Table 4.3 shows the stats of relabel operations for three benchmarks running with CRacer on 16 cores. These numbers are from a single run; general trends are similar across runs. We varied the heavy threshold (described here as the percentage of the total bits in a label — 64 in our case). For the most part, relabels occur infrequently; in fact, we omit the results for `matmul` and `cilksort` because their executions contain no relabels even for very large input sizes. For `heat` and `cholesky`, some relabels occur but not many. We see that the total number of heavy groups decreases as we increase the heavy threshold, but the number of relabels does not change. This makes sense, since with a lower threshold, more groups get marked as heavy early, but no relabel is triggered until some group becomes full.

The interesting one is `fft`, which triggers the most relabels among all our benchmarks. When the heavy threshold is 1 (last row) — that is, we only split full nodes — as expected, most

relabels split only a single heavy group; very occasionally, a couple of heavy groups may become full concurrently and causes a relabel to split more than one group. Even by just decreasing the threshold to be .875, we already see that the median of the number of heavy groups increase to 5.7 and the number of relabels reduces dramatically. This trend continues as we decrease the heavy threshold. This result demonstrates that when many groups can become full during the execution, eager splitting indeed helps in decreasing the number of relabels.

4.5 Related Work

Netzer [137] formalized definitions for determinacy and data races. Static checking for races has been studied in [66, 131, 67], while [46, 138, 5] investigate post-mortem analysis. Research on race detectors for general parallel computations include [72, 164, 152, 65]. In addition to English-Hebrew labeling and SP-Order (and the related SP-hybrid), many algorithms have been proposed for on-the-fly race detection for series-parallel programs. Offset-span labels [132] are shorter than English-Hebrew labels, but still not bounded by a constant. The Nondeterminator race detector [71] runs serially and uses Tarjan's nearly linear-time least-common-ancestor algorithm [178] to simulate the SP-parse tree of a computation. Raman et al. [154] develop a race detection algorithm for async-finish parallelism and implement it in Habanero Java [40]. Raman et al. [155] developed an simple parallel algorithm that maintains the entire computations tree; while it provides good empirical results, it provides no theoretical guarantees. That implementation takes advantage of compiler optimizations to avoid instrumenting all memory access; we have not yet applied such optimizations. Lastly, in contrast to existing race detectors for fork-join computations, TARDIS [105, 127] does

not explicitly keep track of the series-parallel relationships among strands. Instead, it employees a log-based access set and detects races by intersecting the access sets of logically parallel subcomputations at the join point. Moreover, it does not provide provably good time bounds as intersecting the access sets can lead to an asymptotic increase in the span of the computation.

Order maintenance is related to the problem of **online list labeling**. Here, each element in a list / total order must be assigned a distinct integer label $L(x)$ such that $L(x) < L(y)$ whenever x precedes y in the total order. List labeling is sufficient to implement order maintenance. Queries amount to label comparisons. During inserts, some elements may need to be relabeled, e.g., if a new element is inserted between existing elements with consecutive labels. Dietz's first OM structure [58] implements list labeling by way of a weight-balanced binary tree, assigning $O(\log n)$-bit labels in $O(\log n)$ amortized time per insert. The $O(\log n)$ insertion bound is optimal for list labeling using labels of this size [38].

4.6 Conclusions and Future Work

This chapter presented WSP-Order, an on-the-fly algorithm for maintaining series-parallel relationships in fork-join programs. The algorithm operates in parallel without any asymptotic overhead. We validated the theoretical performance by implementing this algorithm together with an access history implementation, yielding CRacer, a practical race detection system with good speedup in practice.

There are undoubtedly many engineering improvements that could be made to CRacer. Since WSP-Order uses quite a bit of extra memory, one natural question is how to recycle memory for nodes in the order-maintenance data structure. Currently, once a strand is inserted it

will never be removed. But there may be times when we can detect when a strand will never be used again and reuse this memory.

WSP-Order is the only known algorithm for series-parallel maintenance that runs both in parallel and asymptotically optimally. Although its asymptotic performance cannot be improved, perhaps there are other asymptotically optimal algorithms that are simpler. Additionally, it would be interesting to consider how to extend this technique to other classes of parallelism, including pipeline parallelism or the use of futures. Chapters 7 and 8 discuss race detection for two such classes, although the technique used is quite different.

```
INSERT(x, y) // executed by worker w
 1  y_inserted = false
 2  repeat
 3       g = GROUP(x)
 4       if IS_FULL(g)
 5            if TRY_ACQUIRE(global_lock)
 6                 // push RELABEL onto DS deque and
 7                 START_RELABEL() // switch to DS deque
 8            else
 9                 JOIN_RELABEL() // switch to DS deque
10       elseif TRY_ACQUIRE(w.local_lock)
11            INSERT_INTO_GROUP(g,x,y)
12            y_inserted = true
13            if IS_HEAVY(g)
14                 ADD_TO_HEAVY_GROUPS(g)
15            RELEASE_LOCK(w.local_lock)
16       else          // local lock not acquired, relabel in progress
17            JOIN_RELABEL() // switch to DS deque
18  until y_inserted = true

RELABEL() // invariant: global lock is held
1  parallel for each worker w
2       ACQUIRE(w.local_lock) // spin until successful
3  Build array H of all heavy groups
4  parallel for each index i of H
5       PARALLEL_SPLIT(H[i])
6  PARALLEL_REBALANCE_TOPLEVEL()
7  parallel for each worker w
8       RELEASE(w.local_lock)
9  RELEASE_LOCK(global_lock)
```

Figure 4.2: Pseudocode for the insert and the relabel procedures of the OM data structure.

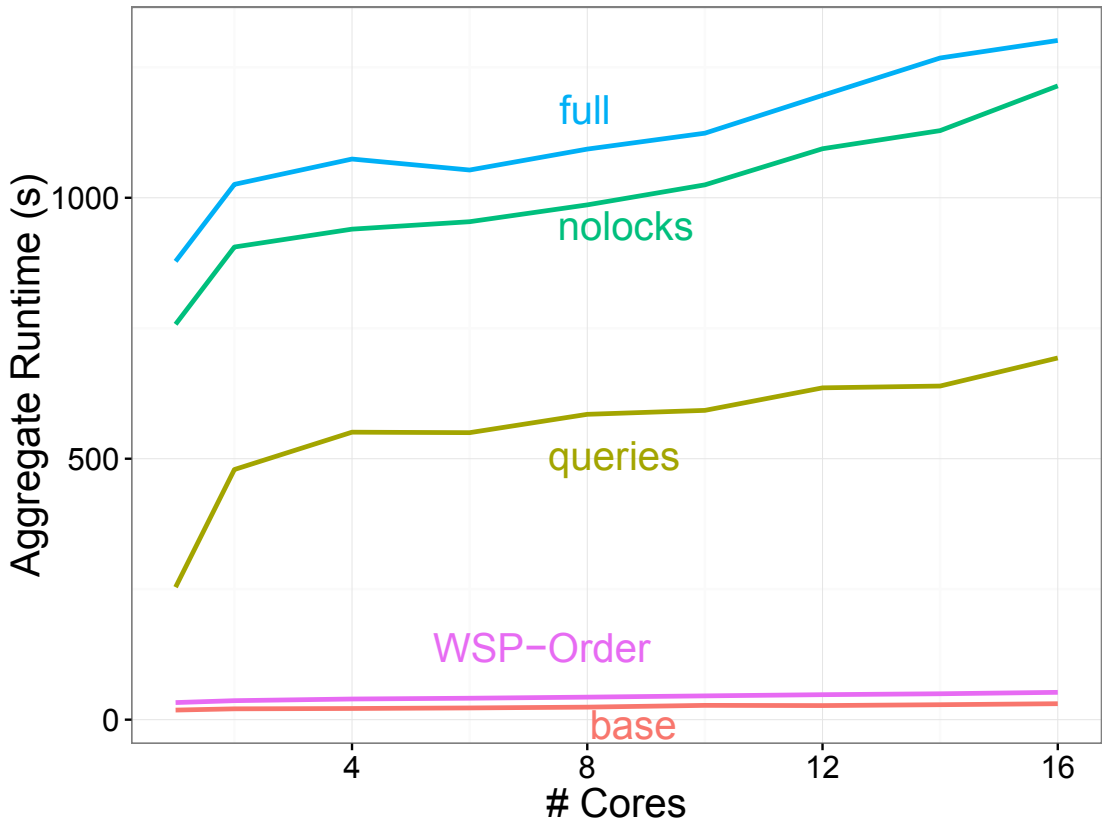


Figure 4.3: Detailed breakdown of CRacer overheads for fft, in seconds. The lines `base` and `inserts` show the overall processing time when running the baseline and inserts configuration described earlier. The line `queries` shows the processing time after including the overhead for queries on top of the inserts configuration. The line `nolocks` shows the processing time of CRacer running in full configuration but does not acquire locks when updating the shadow memory. The line `full` shows the overall processing time of CRacer running in the full configuration.

Chapter 5

Runtime Scheduler Support for Non-Blocking Suspension

The Cilk Plus runtime scheduler is very effective for scheduling fork/join computations. Unfortunately, not all programs can be modeled as fork/join. One major gap includes programs that wait on asynchronous events. These may include external events such as communicating over a network or performing I/O, or internal events involving synchronization objects.

One advantage of static threading is that threads blocked on events (e.g. I/O) do not impede other threads since this model relies on the operating system to schedule threads. In Cilk, however, many lightweight strands are mapped to workers by the runtime scheduler. If the runtime scheduler has no knowledge of active asynchronous events (as is typically the case), a worker executing a strand that waits for an asynchronous event is blocked from doing any work whatsoever. Blocking Cilk workers this way can cause loss of parallelism, deadlocks, and unpredictability.

There is a tension here between programming for concurrency — attempting to minimize response time — and programming for parallelism — attempting to either maximize throughput or minimize completion time. In this chapter we help to alleviate this tension by supporting asynchronous events without real blocking. Strands that need to wait on an asynchronous operation can "suspend," allowing the worker to work on other strands. Strands are resumed when the asynchronous operation has completed (although not necessarily right away).

Two previous runtime systems have attempted to deal with this problem. Concurrent Cilk [192] explores "promoting" some strands to full-fledged operating system threads when waiting on asynchronous operations. The Latency-Hiding Work Stealing scheduler [135] is more similar to our system, though we will see later the different design choices made. Both these systems are specialized for external events; they do not readily apply to the use of synchronization objects. Our runtime can deal with both external events and internal synchronization objects, expanding the class of computations that can be efficiently handled by our dialect of Cilk.

We start with a description of our low-level non-blocking API and then some examples of interfaces that can be built on top of it in sections 5.1 and 5.2. Section 5.3 describes how the runtime system implements the low-level API. Related work is discussed in section 5.4 and section 5.5 concludes.

5.1 Non-Blocking API

We first present our programmer interface for suspending and resuming. This API is fairly low-level and ideally is used by library designers to build higher-level interfaces.

```
1    void cilk_suspend(strand* save); // suspend and go work steal
2    void cilk_resume(strand* save, strand resume); // continuation is resumable
3    void cilk_suspend_and_resume(strand* save, strand resume);
4    void cilk_mark_resumable(strand*);
```

Figure 5.1: The application programming interface that allows non-blocking suspension.

Our terminology here differs slightly from our theoretical analyses of DAGs: in this chapter we use the term **strand** to refer to a chain of nodes with no parallel constructs. A strand suspended by the programmer is either **blocked**, meaning it cannot continue, or **ready**, meaning that is can be resumed but has not yet. All other strands are **active**, even if they are waiting at a sync point.

5.1.1 Suspending

The API functions for suspending a strand are shown in figure 5.1. The simplest function for suspending is `cilk_suspend`, which suspends the current strand and causes the worker thread to begin work-stealing. Strands that need to suspend execution must have some information saved so that they can later be marked as ready to be resumed. This is accomplished be defining an `strand` data structure (opaque to the programmer), and passing in a pointer to a `strand` to each suspension API call. Each suspension-related API call will use the pointer to write information about the suspended strand to that location.

Two other functions are provided that allow the programmer to specify a strand to resume, rather than work-stealing. The `cilk_resume` function suspends the current strand and resumes the strand indicated by the second parameter. The `cilk_suspend_and_resume` function does the same except the suspended strand is immediately marked as `ready`.

It is expected that the programmer initiate the asynchronous operation before making a suspension call. This includes taking any necessary steps to handle notification of operation completion, such as registering in an event loop.

5.1.2 Enabling Resumption

At some point suspended strands must be marked as ready. For example, an event loop may pool all outstanding network operations to handle their completion. A strand is marked ready by calling `cilk_make_ready(strand*)`. Note that we make no prescriptions on how users are notified that an event has completed. This low-level approach is thus applicable to many event-handling approaches, from polling event loops to signal handlers.

5.1.3 Resuming

Once a strand has been marked as ready, the runtime scheduler will automatically ensure that it is eventually resumed, so no action is required by the programmer. However, there may be times when a (newly) ready strand is more important than the strand currently executing on a worker. In this case the programmer can explicitly resume the ready strand by calling `cilk_suspend_and_resume`, as discussed above.

5.2 High-Level Interface Examples

The low-level API described above is really meant for library writers; it requires extra scaffolding to be used in many common cases. We give two examples of high-level interfaces that can be provided using the API. The first deals with internal events – synchronization events called **futures** – while the latter handles asynchronous I/O.

```
1 int foo(int x) {
2     future<int> f = create_future bar(x);
3     future<int> g = create_future baz(x);
4     int y = baz(x);
5     int z = get_future f;
6     return y + z;
7 }
```

Figure 5.2: A simple program demonstrating the parallelism primitives used with futures.

5.2.1 Futures

Futures were first proposed in the late 1970s [76, 14] and have since been implemented in a variety of parallel platforms, including Habanero Java [40], X10 [43], Chapel [41], and C++11 [1].

Since Cilk Plus is an extension of C/C++, futures are available to use in Cilk programs as of C++11. However, creating a future in this manner creates a heavyweight operating system thread, causing a Frankenstein mix of persistent and dynamic multithreading. Using such futures prevents any performance guarantees and results in high overhead when many futures are used. Our system integrates futures into a modified Cilk Plus runtime to provide efficient scheduling of future tasks.

There are two parallel primitives used for programming with futures. Parallelism is created with `create_future`, while parallel computations can be joined with `get_future`. Figure 5.2 shows a simple example using these primitives. The `create_future` primitive precedes a function call to `bar`, promising to deliver the result of `bar` without suspending `foo`. In particular, the statement immediately returns with a **future handle**. In other words, `bar` and the instructions in `foo` after `create_future` can execute in parallel. The `get_future`

primitive can be applied to a future handle to join with the corresponding computation. In figure 5.2, the `get_futureblocks` until `bar` has finished, then returns the result.

`Create_future` operations create parallelism much like a `spawn` does (and would be implemented in a similar manner). However, unlike a `spawn`, futures need not be associated with an explicit `sync` operation. In fact, as figure 5.2 shows, use of futures can generate non-fork/join computation DAGs.

Suspending. Figure 5.3 shows how to handle a `get_future` operation on a future in the form of the `get()` method. We first check if that result is ready. If so, we simply continue on, returning the result. If not, however, we utilize the "suspend" functionality of our runtime system, allowing the current worker to work on other available tasks. The suspended strand is put at the end of a list of strands waiting on the result of that future.

Resuming. When a future computation completes, it must use the `finish` operation to set the result and notify suspended strands that it is complete. Each suspended strand should be made ready to continue with the `cilk_make_ready` call, though we leave it up to the implementer whether one of the suspended strands should be immediately resumed by the worker executing `finish`.

Similarly, many other synchronization objects can be supported, e.g. IVars, MVars, and channels [9]. We point out futures because they are a very common parallel construct in programs today, and we focus on them in later chapters.

```
1  template <typename T>
2  class future {
3  private:
4    T data;
5    int status;
6    strand_list suspended_list;
7
8  public:
9    void finish(T result) {
10     data = result;
11     status = 1;
12     // mark all suspended strands as ready
13   }
14   T get() const {
15     if (status == 0) {
16       // append new strand node to suspended_list
17       cilk_suspend(&node);
18     }
19     return data;
20   }
21 };
```

Figure 5.3: Example pseudocode for implementing futures using our non-blocking suspension API.

5.2.2 Asynchronous I/O

Handling asynchronous I/O is slightly trickier, since external events complete at arbitrary times, unrelated to the computation itself. One approach to such a library is to provide an event loop for handling I/O events. This event loop might run in a kernel thread in the background, for example, polling for completion of various I/O operations.

At an I/O operation, an event is registered with the event loop and the current strand is suspended. When the event loop notices that an operation has completed, it marks the strand as ready to be resumed.

5.3 Design of the Runtime Scheduler

This section discusses the runtime scheduling changes necessary to support suspending and resuming strands. The notion of suspended strands is not completely foreign to the Cilk Plus runtime system. For example, consider a computation that spawns a long-running task. In the meantime the continuation is stolen and completed, but cannot continue past the sync point until the long-running task is completed. At this point the continuation strand is suspended and the worker resumes work-stealing.

However, because of its restriction to fork/join computations, Cilk Plus is guaranteed that any strand needs to be suspended only when that worker's deque is completely empty. This is not true for our runtime, which may suspend in arbitrary places.

There are three main design points in such a runtime, which we discuss in turn: suspending strands, marking suspended strands as ready, and resuming ready strands. We will explain

each in turn, followed by discussions on how they affect stealing policy and memory use, respectively.

5.3.1 Suspending Strands

There are two basic design choices for suspension. Conceptually, the simplest approach is to suspend the current strand and simply continue working from the bottom of that worker's deque. Unfortunately, this is difficult to because of Cilk Plus' implementation [1].

An alternative approach, and one our scheduler takes, is to suspend the entire current deque! This approach requires that each worker have a dynamically changing number of deques. This may seem drastic compared to the simpler approach, but it is really a generalization of the runtime systems presented in the previous chapters, which allowed each worker to have two deques. Suspending is then as simple as using C's `setjmp` facility and saving this context information with the suspended deque. In fact, because of this design choice, the `strand` data structure from the previous section is simply a pointer to the deque structure, which must additionally contain the context information from `setjmp`.

To support these suspended deques, each worker now keeps two **deque pools** of suspended deques, one for blocked deques (containing blocked strands) and another for ready deques (containing ready strands). For simplicity, these deque pools are simply resizable arrays of deque pointers.

[1] For those familiar with the Cilk Plus runtime system, it is difficult to "promote" the bottom frame in a deque. However, this approach is certainly possible with some engineering effort.

5.3.2 Resuming Strands

When marking a blocked deque/strand as ready, we need to transfer the deque from a worker's blocked pool to its ready pool. To avoid having "holes" in a deque pool, we keep a index `tail` denoting the largest used index in the pool. When removing a deque, we decrement the tail and swap the deque at the old tail with the deque we are marking as ready. We are then free to add this ready deque to the pool of ready deques.

Because of our design choice to suspend entire deques, resuming a ready deque is relatively simple. We remove the deque from the ready pool, replace the worker's deque with it, then use the context information stored with it to `longjmp` to it. Of course, if we are marking a deque as ready and going to immediately resume it, there is no need to place it in a ready pool. In that case we skip that step and directly resume the deque.

Note that our API provides a choice here: what should we do when a worker marks a strand as ready? Do we simply mark it as ready (moving the blocked deque to a ready pool), or should we suspend the current strand and jump to this newly ready strand? The right decision here depends on the application, so our API does not force either policy.

5.3.3 Stealing policy

For work-stealing to be efficient, *all* ready strands must be "stealable," even those on the deque of a suspended strand. Hence we must change the stealing policy to choose among all deques, rather than just the P active deques that exist in vanilla Cilk. One approach for this problem is to keep a global array of all deques in the system, as in Muller and Acar's runtime [135]. We chose worker-local deque pools to avoid the synchronization required for such a global structure.

To perform a steal in our system, a thief first chooses a victim worker, then selects among their deques (active, suspended, and ready) to find a victim deque. If we select a ready deque, we **mug** the deque – taking it as our own, and using the associated context information to `longjmp` to the suspended strand. If we select a suspended deque with no work, we remove the deque from the suspended pool of that worker. Such a deque will not be touched until it is either marked as ready or directly resumed.

While relatively simple to implement, this approach has a problem. Work-stealing only load-balances effectively if all workers have approximately the same number of deques. We can take a randomized approach to balancing the number of deques, just like work-stealing itself. Whenever we suspend a deque, we choose two workers at random and transfer a suspended deque from the worker with fewer deques to the worker with more. As chapter 6 shows, this approximately balances the suspended deques among the P workers.

5.3.4 A Note on Memory Use

Every time our runtime suspends a strand (and therefore a deque), it must also allocate a new runtime stack to use for any new work that will be stolen. A natural concern here is exorbitant memory use. Fortunately, though plenty of *virtual* memory is used, few physical pages are allocated. We found in chapter 6, for example, that although the number of memory mappings was very high, very few pages were touched on each runtime stack.

5.4 Related Work

The most related work is the latency-hiding work stealing runtime scheduler by Muller and Acar [135]. That runtime has the same goals as ours, though they make different design choices. For example, a deque is not necessarily suspended when a strand is suspended,

though strands are still "attached" to deques – if a deque is empty but still has suspended strands attached to it, the deque is suspended. That runtime is also focused on handling external asynchronous events such as I/O, tying itself to an event loop to handle event completion. More importantly for the purposes of this dissertation (and the parallel computing research group here at WUSTL), that runtime is implemented in parallel ML, while our runtime is implemented in Cilk Plus, making it usable with chapters 6 to 8 and perhaps other projects.

As discussed in the introduction, another approach is to "promote" blocking strands to operating system threads. Though sufficient for some application, this creates a complicated interplay between the operating system scheduling such promoted threads and the Cilk worker threads, making is harder to reason about system performance. Other approaches to providing lightweight non-blocking tasks either changes the C calling conventions, breaks legacy support [17], or creates a full runtime call-stack for each task [188].

The proof of work-stealing in Arora, Blumofe, and Plaxton [8] does not rely on any particular structure of the computation DAG, so it can apply to internal synchronization events such as futures if they are handled correctly. However, that bound would require working from the bottom of the deque when a strand is suspended. As noted, doing so in Cilk Plus is difficult for technical reasons. Moreover, we prove a bound for this design in chapter 6, as well as verifying that our scheduler is empirically efficient.

5.5 Conclusions and Future Work

This chapter presented a runtime scheduler for a broader class of computations than fork/join, encompassing both use of internal synchronization events and asynchronous external events.

The system was actually developed to support the research in chapter 6, but once developed we realized its value as a standalone system. The code for the system is open source at available at www.gitlab.com/wustl-pctg-pub/mdcilk.

It is ongoing work to evaluate the scheduler as a standalone system. What applications will see improved throughput or runtime with the ability to utilize all forms of parallelism, rather than just fork/join? What applications will see improved responsiveness by avoiding blocking for external events? It would also be interesting to compare this system to the two related works previously mentioned: Concurrent Cilk and Latency-Hiding Work Stealing, though the latter would require significant engineering effort to port to Cilk Plus.

There are many possible design variants for such a system. For example, we are in the process of developing a system that avoids the need to suspend entire deques when a strand is suspended. We suspect that some applications will see improved performance with this variant, though some applications are likely better off with the current design.

Chapter 6

Processor-Oblivious Record and Replay of Lock Acquisitions

In programs that use locks to protected shared objects, critical sections protected by the same lock execute in non-deterministic order. This is caused by different thread interleavings producing different lock acquisition orders. This non-determinism makes reproducing bugs notoriously difficult — a bug that manifests under one interleaving may not appear under another.

```
1     cilk_for (int i = 0; i < pq.size(); i++) {
2       Element e;
3       lock(p);
4       e = pq.removeMin();
5       unlock(p);
6
7       lock(l);
8       list.append(l);
9       unlock(l);
10    }
```

Figure 6.1: A DRF program with an atomicity bug.

Consider the simple Cilk program in figure 6.1, where multiple tasks remove the minimum element from a priority queue and add it to a list. The goal of this program is to produce a sorted list of numbers. However, it contains an atomicity bug: because the accesses to the priority queue and the linked list are in separate critical sections, items may not be added to the linked list in order. Nevertheless, the bug only manifests if the tasks interleave in a particular manner. A record and replay system could help identify this bug by deterministically replaying a buggy execution, allowing a programmer to determine that two items are removed from the priority queue before either is added to the linked list.

A popular technique for addressing such non-determinism is **record and replay** [189, 149, 97, 116, 186, 33, 63, 82, 125, 190, 146, 136, 157, 118, 113, 96]. One execution *records* enough information about its behavior so that a second execution can faithfully *replay* that behavior, producing the same outcome. As a result, any bug that manifests during the recorded run will be reproduced during the replay run, easing the task of tracking down bugs.

In particular, this chapter focuses on programs that protect shared objects with locks. A record and replay system for these programs must ensure that critical sections protected by the same lock are executed in the same order during the record run and the replay run. Prior work on record and replay generally records *thread* interleaving, and tracking the behavior of the threads of a program as they execute, ensuring that during replay, threads interleave in the same way when executing critical sections. While this approach succeeds at its goal of replaying recorded behavior, it has the drawback of requiring that the replayed run use the same number of threads as the recorded execution.

This concession seems mild for static multithreaded programming models, where the number of threads is known. However, in a dynamic multithreading model the number of threads is not part of the model – the program is not even aware of how many processors or threads

are active. In the model described in chapter 2, programmers express logical parallelism in the program using the primitives spawn and sync and parallel-for loops, while a scheduler efficiently maps the parallelism to worker threads at runtime.

We are aware of no existing record and replay systems for dynamically multithreaded programming models. Even when keeping the same number of workers for recording and replaying, existing record and replay systems would face high overheads if applied to dynamic multithreaded programs. This is due not only to the many fine-grained tasks involved but also due to scheduling decisions. Dynamic multithreaded programs are usually scheduled with a randomized work stealing runtime scheduler: *which* workers execute *which* tasks *when* is non-deterministic and changes from one run to another. Including this non-determinism in a record-and-replay system would dramatically increase the information necessary to record.

This chapter presents PORRidge, a record and replay system designed for dynamically multithreaded programs. Unlike previous systems, this system is process-oblivious, just like the dynamic multithreaded model. This allows a program recorded using n worker threads to be replayed on m worker threads, where m may be greater than n!

PORRidge makes the following assumptions about programs. First, we assume that there are no parallelism constructs within critical sections. This is a standard assumption for dynamic multithreaded systems. We also assume programs are data-race free – the order of synchronization operations is controlled, given that accesses to shared data is correctly synchronized. While this seems like a strong assumption, we note that this is a common assumption for record and replay systems [82, 157] and that other race detection tools can be used before PORRidge is applied to a program.

Further, this assumption is due to our *implementation* only, which needs to track sources of non-determinism. The same techniques could apply to racy programs by using other tools to indicate any additional sources of non-determinism, such as determinacy races.

Contributions

In return for these assumptions, PORRidge provides the first record and replay system designed specifically for dynamically multithreaded programs. It is processor-oblivious and both theoretically and empirically efficient. By not recording runtime scheduling decisions PORRidge is able to achieve nearly optimal performance guarantees for recording and replaying. In particular, consider a computation with T_1 work and T_∞ span (as usual). Since we are now considering a program with locks, let B be the amount of work in critical sections. Then PORRidge records a computation in time $O\left(T_1/P + T_\infty + B\right)$ with P workers. For a single lock, this bound is *asymptotically optimal*. Replay incurs higher overhead due to the need to respect additional dependencies on the order of critical section, running in time $O\left(T_1/P' + T'_\infty \log\log P'\right)$ on P' workers, where T'_∞ is the span of the DAG augmented with the additional happens-before edges[1]. The consequence is that by using more cores we can make replay *asymptotically faster* than the recorded run.

We implemented our design using the Cilk Plus [99] runtime system presented in chapter 5. Our system shows good scalability and low overhead over a suite of six benchmarks, despite the need to respect extra dependencies during replay.

Outline. Section 6.1 describes the design details that allow PORRidge to efficiently record and replay lock acquisitions. We prove and discuss the performance bounds in section 6.2.

[1]Do not confuse this term with the augmented span term from chapter 3; we use the same term but different notation here.

We present our implementation and empirical evaluation in section 6.3 and conclude in section 6.5

6.1 Design of PORRidge

The key insight behind PORRidge is as follows: there are multiple sources of non-determinism in scheduling when we execute a dynamic multithreaded program. For example, the random work stealing decisions that the scheduler makes. However, for a data-race free computation, a recording run need not record all this information to reproduce it faithfully during replay. The DAG that represents a computation is independent of the number of workers and for race-free computations, the only non-determinism arises from the order that tasks acquire locks. These lock acquires represent additional **happens-before** edges in the program DAG and recording these additional edges, i.e. the order in which various critical sections acquired a shared lock, is sufficient to ensure that the DAG can be replayed faithfully.

Therefore, during a recording run, PORRidge simply records these happens-before edges. More importantly, during the replay run, PORRidge ensures that the happens-before relationships that were recorded are respected: in other words, during replay, PORRidge schedules the **augmented DAG** which contains all these happens-before edges in addition to the original dependencies. While this new augmented DAG may have parallelism limited by the happens-before edges, its parallelism is *not* directly limited by the number of threads that the recording run executed on.

Another important property of PORRidge is that the recording system sits *on top* of its runtime. One possible way to record a Cilk computation is to include the Cilk runtime

in the scope of what is recorded, recording and subsequently replaying all of the non-deterministic decisions regarding work stealing. But the Cilk runtime is highly parallel and non-deterministic, and including it in the recording scope would dramatically increase the amount of information to be recorded. Instead, PORRidge only records the happens-before relationships.

Replay is more complex. The Cilk runtime system is not designed to obey happens-before edges that are not directly part of the program itself. Therefore, PORRidge uses the mechanisms from the runtime system in chapter 5 to respect these dependencies. However, these mechanisms, and generally all of the non-determinism of the scheduler, remain encapsulated separately from the replay itself. By keeping the runtime (both during record and during replay) outside the scope of the system, PORRidge is able to maintain low overhead.

6.1.1 Recording

PORRidge provides wrappers for the various thread lock objects and associated acquire/release functions. Conceptually, a lock object in PORRidge contains a pointer to the underlying lock defined by the POSIX pthread specification [98] and an ordered list of successful lock acquires to this lock. When a worker successfully acquires a lock, it simply adds its currently executing strand (in the form of a *strand ID*, defined below) to the end of the list. If the lock is not available, the worker spins. At the end of the recorded execution, every lock object writes out the strands in the list to a log file in the order inserted.

Identifying Strands. For processor-oblivious replay, the information stored in the list must uniquely identify the strands in the computation DAG and the identification must be consistent across executions. Here, we use the idea of a **pedigree** [124], a sequence of integers

corresponding to the rank ordering of spawn statements in the ancestor functions (including this function) that lead to the current strand. Pedigrees uniquely identify each strand in a consistent manner since it depends only on the computation DAG and not on the schedule of strand execution.

The open-source Cilk Plus runtime [99] readily provides support for pedigrees; however, each read to a pedigree incurs a worst-case $\Theta(d)$ overhead, where d is the maximum **spawn depth**, the number of spawn statements nested on the stack during serial execution. Since the pedigree must be read in every lock acquire, this causes lock acquires to incur $\Theta(d)$ overhead during record and replay. Ideally, we would like to keep the cost of lock acquire to be constant in order to guarantee both the record and replay time bounds.

To achieve the desired constant overhead, we use a strategy similar to DotMix [124] to give each strand a **strand ID**, which is effectively a hash of a pedigree that can be maintained and derived with constant overhead. DotMix works as follows. The runtime generates a size-d vector [2] of random numbers using the seed at the beginning of the computation. Given a pedigree, DotMix takes dot-product of the pedigree with the vector and mods the dot-product result with a large prime p; a pedigree always hashes to the same random number provided that we use the same seed. Moreover, two random numbers generated via two different pedigree have a low probability of collision [124]. Using a similar strategy as DotMix, we obtain unique strand IDs with constant overhead per lock acquire. In particular, we calculate this dot-product online, adding constant overhead at each spawn, sync, and function return to maintain the correct, active strand ID for each worker.

[2] In practice, we do not know d ahead of time, so we simply use a constant (256) known to be large enough to support any reasonable computation.

Storing Strand IDs. The drawback of strand IDs is the (rare) possibility of collision. To detect collisions efficiently, the runtime employs a **hash-list** (instead of a list), which is a hash table whose values form a list implicitly by adding a next pointer to each entry in the hash table. To disambiguate collision, whenever a worker tries to add a strand id which is already in the table, it marks the first strand with the same strand ID as "has collision," reads the full pedigree of the current strand and stores it in the hash-list. Note that a hash-list is kept separately for each lock. Finally, during recording, the runtime also keeps track of the head and the tail of the list with a lock object — the tail so that it can manage the implicit list and the head so that it can write the result to the log at the end of the execution.

6.1.2 Replaying

At the beginning of the replay, the runtime reads in the previously recorded log and recreates the hash-list. We maintain the invariant that the head pointer to the list node always points to the next strand that should successfully acquire the given lock. Each list node also contains a pointer to the runtime data necessary to enable suspending and resuming the strand (the strand data structure described in chapter 5). During replay, if a worker encounters a lock acquire for critical section a, and its predecessor — a lock release of the critical section b that was executed immediately before a during the recording run — has not executed yet, the worker should suspend the execution of the strand, since it is not ready in the augmented DAG. On the other hand, when some worker (in this case, the worker that executed b) releases a lock, it may enable critical section a (which was earlier tried and suspended). This worker must then resume this suspended critical section.

Lock Acquire. When a worker encounters a lock acquire, it checks the head pointer to see if this is the strand that should get the lock next. If so, it acquires the lock and continues

execution. Otherwise, the worker hashes its strand id and marks the corresponding hash-list node to indicate that the corresponding strand has been tried and suspended, and suspends the execution. The worker then uses the suspend capability of chapter 5 to suspend the strand and starts work stealing. Note that a worker cannot spin wait on a lock acquire that is not ready since this can lead to deadlocks.

Lock Release. To release a lock, the releasing worker first advances the head pointer and checks to see if the next strand has been tried and suspended. If not, the worker simply continues the execution after the lock release. If the next strand has been tried and suspended, the worker performing the lock release now has two continuations that it can potentially work on — the continuation after the lock release, and the suspended lock acquire enabled by this lock release. Both choices lead to the same theoretical guarantees. In our implementation, we chose to have the worker suspend the continuation after the lock release and resume the next lock acquire in the list to reduce contention. Note that it is possible for a worker to release a lock while a different worker is concurrently suspending the next strand in line — the synchronization is coordinated using a Dekker-like protocol [59], since there are at most two workers concurrently operating on a given list node.

Handling Strand ID Collisions. A worker may encounter a head list node marked as "has collision," — in this case, multiple strands with the same id acquired this lock. Recall that during recording, the runtime stores full pedigrees only when it discovers a collision; therefore, the first strand involved in the collision has the strand id stored and the remaining strands involved in the collision have full pedigrees stored. Thus, if a worker finds a head node with the same strand ID but marked as "has collision," it must read its full pedigree and compare it against other list nodes hashed with the same strand IDs (for which the full

pedigree is stored). If none of them match the current pedigree, it can proceed with getting the lock. Otherwise, it must suspend.

6.1.3 Runtime Modifications

We originally designed and implemented the runtime system in chapter 5 for this project, though its functionality is broader. The fact that a lock acquire causes a worker to suspend its current execution causes the PORRidge scheduler to diverge from the vanilla work-stealing scheduler used by Cilk Plus without locks. As with the previously presented schedulers, the PORRidge scheduler no longer maintains the P-deque invariant of vanilla work stealing since worker can suspend execution when it has a non-empty deque. Thus the runtime must handle multiple deques per worker, and additional care must be taken to provide the provably-scalable time bound for replay.

During replay, a worker can suspend execution (1) upon a lock acquire if the lock acquire is not ready, or (2) upon a lock release, if the lock release in turn enables a suspended lock acquire. In the first case, if the worker suspends its current (non-empty) deque, work steals and allocates a new deque for the stolen work, thereby increasing the total number of deques. In the second case, the worker suspends the continuation of the lock release, and resumes the deque containing the lock acquire that it just enabled; in this case, the overall number of deques in the system does not increase. Note that the deque redistribution policy in chapter 5 is necessary to balance the load of suspended deques among the workers; we use this fact to provide the provably-scalable replay time bound in section 6.2. The mugging of entire suspended deques on steals (also described in chapter 5) is not necessary for this time bound, but improves performance in practice.

6.1.4 Performance Optimization

Thus far we have been discussing the design assuming that the lock-acquire ordering for a given lock is represented using a hash list. The hash-list representation works, but it can incur large overhead in practice for benchmarks that are already memory-bound (such as the graph benchmarks described in section 6.3), since random accesses to the hash list inherently lack locality and incur additional cache misses. We optimized the implementation of the record phase in PORRidge by using a small bloom filter to detect strand ID collisions in place of a hash table. Doing so allows the PORRidge to store the bloom filter with the lock object itself, leading to better spatial locality, and it uses much less space than keeping an actual hash list. The trade-off is that a bloom filter can report false positives (i.e., detecting collisions between strand IDs with different values) and thus may lead to reading and logging the full pedigrees unnecessarily. In our experiments, however, we found that using the bloom-filter outperforms the hash list due to cache effects.

Even though we were able to use a bloom filter during recording, the same optimization does not work during replay, since a worker uses the hash table not to identify collisions but rather to find the list node corresponding to the encountered lock acquire quickly. During replay, a worker encountering a lock acquire that is not yet at the head of the list needs to find the corresponding list node in order to mark it suspended. If the corresponding strand ID is marked to have collision during recording, the worker also needs to search through the list nodes with the same strand ID to precisely identify the correct list node. A bloom filter is not sufficient for these purposes. Nevertheless, for many benchmarks, the number of lock acquires for a give lock is small; thus, keeping the lock-acquire ordering in a simple array and searching through the array suffices. For such benchmarks, the spatial locality and decrease in memory usage when using a simple array outweighs the benefit of constant-time search via

a hash-list. Since the number of lock acquires per lock is known at the beginning of the replay, in our implementation, we optimized the replay to choose between the two representations — PORRidge keeps the lock-acquire ordering in a simple array if the number is small, and it uses a hash-list otherwise.

6.2 Performance Bounds

6.2.1 Record Running Time

The recording process requires no changes to the runtime system and adds only constant-time overhead[3]. We model locks by letting B_i be the total amount of time that lock ℓ_i is held. Then we say that the **total blocking time** is $B = \sum_i B_i$. We first divide the computation into two types of phases. We will bound the time spent in each phase separately.

A phase is **non-blocking** if no processor is waiting on a lock, otherwise it is **blocking**. The following lemma comes from the fact that the total time any processor could be holding a lock is at most B.

Lemma 6.1. *The total amount of time spent in blocking phases is at most B.*

Lemma 6.2. *The total expected time spent in non-blocking phases is $T_1/P + O(T_\infty)$. The time spent in non-blocking phases is $T_1/P + O(T_\infty + \lg 1/\epsilon)$ with probability $1 - 1/\epsilon$.*

Proof. During non-blocking phases, the processors are either working or stealing. The total number of work steps is at most T_1, since each work step consumes a unit of work in the computation DAG. From an argument very similar to that in ABP [8], one can show that the total number of steal steps when no worker is blocked is $O(PT_\infty)$ in expectation and

[3]For ease of exposition we ignore the small probability of strand ID collisions, causing non-constant overhead to read the full pedigree.

$O(PT_\infty + P \lg 1/\epsilon)$ with probability at least $(1 - 1/\epsilon)$. Since there are a total of P processors executing these work or steal steps, the total time spent on non-blocking phases is as stated. Note that some work may also be done during blocking phases; however, this only overestimates the running time. □

Combining lemmas 6.1 and 6.2 yields the following theorem:

Theorem 6.3. *Given a computation with work T_1, span T_∞, and blocking B, if we record the computation on P processors, the running time is $O(T_1/P + T_\infty + B)$ in expectation.*

6.2.2 Replay Running Time

Analyzing the replay process is trickier. When a recorded computation is replayed, the runtime must satisfy additional happens-before edges. These edges are not specified in the original DAG, as a lock can be acquired by multiple threads in any order. We call this new DAG an **augmented DAG**. The augmented DAG has the same work as the original DAG but may have a larger span. We denote the span of an augmented DAG as T'_∞. We will show the following theorem:

Theorem 6.4. *Given an augmented DAG with work T_1 and span T'_∞, the replay process completes in expected time $O\left(T_1/P' + T'_\infty \log \log P'\right)$.*

Note that this bound is nearly tight, since T_1/P and T'_∞ are lower bounds for replaying. The additional factor $\lg \lg P$ is tiny for most machines. Further, for computations with sufficient parallelism, the T_1/P term will dominate, meaning that replay overhead will be minimal.

As with the analysis of recording, we divide time steps into work steps and steal steps. No worker ever waits on a lock, so there are no blocking steps[4] The total work is still bounded by T_1. Therefore, it only remains to bound the number of steal attempts.

We will use the ideas from the ABP analysis [8] to bound the number of steal attempts. The main difference between vanilla work stealing and our replay strategy is that we now have more than P deques. In particular, the high-level idea in the ABP analysis is the following. If there are X deques in the system, then X steal attempts are likely to reduce the critical path by a constant amount. Therefore, the total number of steal attempts is $X \times T'_\infty$ in expectation. Since our scheduler can have an arbitrarily large number of deques (as large as the number of critical sections in the program), we would get a very bad bound if we directly applied that technique. We use additional insights to bound the number of steal attempts for a replay scheduler. We first make the following observation, due to the stealing policy described for our runtime in chapter 5.

Observation 6.5. A steal from a suspended deque always succeeds since it is never empty. Since a successful steal is followed by a unit of work by the thief, the total number of steals from suspended deques is bounded by T_1.

Note also that when the number of suspended deques is small, i.e. still on the order of $O(P)$, we can use an analysis similar to ABP to bound the steal attempts. We only run into issues when the number of suspended deques is not small.

A **work-bounded phase** begins when at least $P/2$ workers have at least one suspended deque. During a work-bounded phase, about a quarter of the steal attempts are likely to succeed (since that many of the steals occur from a suspended deque). Thus we can bound

[4]Again we ignore the negligible probability of strand ID collision which force workers to read their full pedigree. In practice, most benchmark runs see no collisions at all.

the total number of steal attempts in these phases by the work of the computation. A **steal-bounded** phase begins with fewer than $P/2$ workers having any suspended deques. Recall, as described in section 5.3.3, we try to keep the number of deques across workers roughly balanced by throwing deques to workers at random. Therefore, if fewer than $P/2$ workers have suspended deques, the total number of deques in the system are likely to be small. Therefore, we will bound the steal attempts occurred during steal-bounded phases using analysis similar to that in ABP. Note that a phase is either work-bounded or steal-bounded.

Lemma 6.6. *The expected number of steal attempts during work-bounded phases is* $O(T_1)$.

Proof. In work bounded phases, at least $P/2$ processors have suspended deques. Since a thief chooses a victim uniformly at random, we have $1/2$ probability of stealing into these processors with suspended deques. In addition, since these workers have at most one active deque and at least one suspended deque, about half of the steals from these workers are expected to be successful. Therefore, the expected number of steals attempts during work-bounded phases is $4X$ where X is the number of steal attempts from suspended deques. Combining with Observation 6.5 gives the lemma. □

We now consider bounding the steal attempts in steal-bounded phases. Although we now potentially have more than P deques, we can still use analysis similar to APB to bound the steal attempts. At a very high level, the APB analysis works as follows. The computation starts out having bounded "potential," which is a function of the computation's span. Note that the important node that one needs to execute in order to make progress on the span always sits on top of some deque. The key point in the ABP analysis is that, if there are X deques in the system, and we steal uniformly at random from them, then after $O(X)$ number of steal attempts, some worker steals and executes the important node at the top of

some deque and thus make progress on the span. Hence we can bound the number of steal attempts to be $O(XT_\infty)$ in expectation.

Similar to ABP, we define a potential function based on the depth of nodes in the augmented dag. The depth of a node $d(u)$ is recursively defined as 1 plus the maximum depth of all its parents. The weight of a node is $w(u) = T'_\infty - d(u)$. Then, we define a potential as follows:

Definition 6.7. The **potential** $\Phi(u)$ of a node u is $3^{2w(u)-1}$ if u is assigned, and $3^{2w(u)}$ if u is ready.

The total potential of the computation is the sum of the potentials of all its ready and assigned nodes, and the follow lemma follows from the APB analysis in a straightforward manner.

Lemma 6.8. *The initial potential is* $3^{2T'_\infty-1}$ *and it never increases during the computation.*

The following lemma is a straightforward generalization of Lemmas 7 and 8 in ABP [8].

Lemma 6.9. *Let* Φ_i *denote the potential at time* t *and say that the probability of each deque being a victim of a steal attempt is* at least $1/X$*. Then after* X *steal attempts, the potential is at most* $\Phi(t)/4$ *with probability at least* $1/4$*.*

The following structural lemmas follow in a straightforward manner from the arguments used throughout the ABP paper [8], so we state them without proof here.[5] In ABP, X would be P. In our case, we need to analyze what X is. To bound X, we define the number of suspended deques a worker has as its **load**, and we are concerned with the **maximum load**, i.e., highest number of suspended deques a worker can have. We consider two scenarios. First

[5]ABP does not explicitly capture these three lemmas as claims in their paper — some of their proof is captured by "Lemma 8" and "Theorem 9" of [8], but the rest falls to inter-proof discussion within the paper.

scenario is where there are at most $2P$ suspended deques in the system, and we can bound the maximum load in this case.

Lemma 6.10. *Say there are at most $2P$ suspended deques over all processors. With probability at least $1 - 1/P^2$, the processor with the largest load has at most $k = \lg\lg P + O(1)$ suspended deques.*

Proof. The lemma follows from the Azar et. al's [11, 13] classic balls into bins results. They prove that if we throw K balls into P bins by checking two bins and throwing the ball into the less loaded bin, then the maximum load is $\lg\lg P + O(K/P)$ with high probability. That is, the loads in bins are mostly balanced within an additive factor of $\lg\lg P$. If we think of suspended deques as balls and processors as bins, by performing the load balancing of suspended deques described in section 5.3.3, this result guarantees that when deques are suspended, they are distributed evenly. □

We will say that a distribution is **balanced** if the processor with the largest number of deques has fewer than $2\lg\lg P$ deques, otherwise, we will say that the distribution is unbalanced. We must also worry about imbalance creeping in as processors steal from suspended deques and suspended deques disappear. However, the proof of Theorem 4.1 from Azar et. al's paper [13] implies that if we start from an imbalanced distribution, and on each step, pick two random bins, and move a ball from the more loaded bin to the less loaded bin, then after $P^2 \lg\lg P$ steps, bins will be balanced again. Since our strategy for steals from section 6.1 follows exactly this strategy, the next claim follows:

Claim 6.11. *If we start from a balanced distribution, it becomes imbalanced after $P^2 \lg\lg P$ steal attempts with probability at most $1/P^2$. If the distribution becomes imbalanced, it becomes balanced again after $P^2 \lg\lg P$ steal attempts with probability at least $(1 - 1/P^2)$.*

In the other scenario, where there are more than $2P$ suspended deques in the system, we cannot readily bound the maximum load, but one can show that such a scenario falls under the work-bounded phase with high probability:

Lemma 6.12. *Say there are more than $2P$ suspended deques. At least $P/2$ workers have at least one suspended deque with probability at least $1-(e/8)^{P/2} \geq 1-1/P^2$ for large enough P.*

Proof. The probability that $P/2$ workers have no suspended deques is $\binom{P}{P/2}(1/2)^{2P} \leq (2e/16)^{P/2}$. □

We will divide each steal bounded phase into rounds with $2P \lg\lg P$ steal attempts. We say that a round is **good** if the maximum load is at most $2 \lg\lg P$ throughout the round and bad otherwise. Then in a good round, we can generalize the lemma from APB (stated as lemma 6.9), and show:

Lemma 6.13. *Let $\Phi(t)$ denote the potential at the beginning of a good round. After $P \lg\lg P$ steal attempts, at the end of the round, the potential is at most $3\Phi(t)/4$ with probability at least $1/4$.* □

Proof. There are at most $2P \lg\lg P$ deques during the round. Therefore, the probability that a particular steal attempt hits a particular deque is at least $1/(2P \lg\lg P)$ (it may be higher since some workers have fewer than $\lg\lg P$ suspended deques). Therefore, we can apply a small modification to lemma 6.9 generalized from ABP and argue that the total potential decreases. □

Lemma 6.14. *The total number of good rounds is $O(T'_\infty)$ in expectation.*

Proof. Similar arguments to ABP. At a high level, from lemma 6.13, a constant number good rounds suffice to decrease the potential by a constant factor in expectation. Therefore, the number of rounds needed to reduce the potential to one is log of the initial potential, which is $3^{2T'_\infty}$. Therefore, after $O(T'_\infty)$ rounds, the potential disappears and the computation completes. □

We still need to bound the number of bad rounds, however.

Lemma 6.15. *The number of bad rounds is $O((1/P)X)$ where X is the number of good rounds.*

Proof. A round is good with probability at least $1 - 1/P^2$ from lemma 6.10, Claim 6.11, and lemma 6.12. If we ever get into a bad round, things become balanced again after $O(P^2 \lg \lg P)$ steal attempts, or $O(P)$ rounds from Claim 6.11. Therefore, it takes P^2 good rounds before a bad round occurs and then there can be at most P bad rounds before a good round occurs again. □

The following lemma follows from lemmas 6.14 and 6.15 and the fact that each round has $P \lg \lg P$ steals.

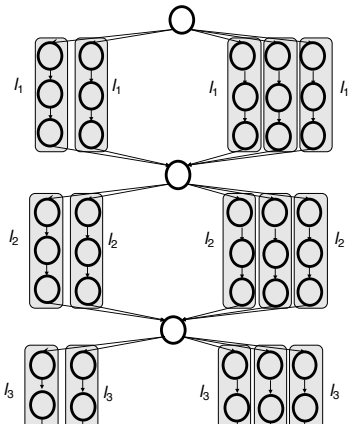
Lemma 6.16. *The expected number of steal attempts in steal bounded phases is at most $O(T'_\infty P \lg \lg P)$.*

The following lemma follows from lemmas 6.6 and 6.16.

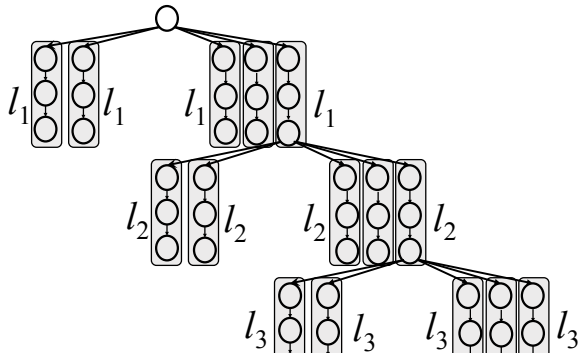
Lemma 6.17. *The total number of steal attempts across all phases is $O(T_1 + T'_\infty P \lg \lg P)$.*

Lemma 6.17 and the fact that the total amount of work is T_1 implies theorem 6.4.

6.2.3 Discussion



(a) An example DAG with multiple locks where the recording bound is tight since all critical sections must execute sequentially.



(b) An example DAG with multiple locks where getting a tight bound for recording is impossible for an online scheduler. The offline scheduler can always schedule the "important" (in this case, the right-most) critical section first, but an online scheduler has no way of knowing which critical section is "important", and therefore may execute it last.

Figure 6.2: Examples for bad DAGs with multiple locks.

We now discuss how good or bad these bounds are, theoretically. For a single lock, note that T_1/P, T_∞, and B are all lower bounds on the execution on P workers; therefore, the bound is tight. For multiple locks while T_1/P and T_∞ are still lower bounds, B is not a lower bound for all dags. Nevertheless, this bound is existentially tight — there exist dags for which it is tight. Consider the DAG shown in figure 6.2a. In this dag, the gray rectangles represent critical sections and critical sections marked l_i all access the same lock; similarly for lock l_2. Imagine that this dag continues on and each layer accesses a different lock. In this dag, all the critical sections must execute sequentially after one another; therefore b is a lower bound. Therefore, for multiple locks, our bound is existentially tight; that is, there exist dags for which this bound is tight.

An existentially tight bound may not be satisfying. In general, however, it is difficult for online schedulers to get tight bounds on all computation DAGs with multiple locks without knowing what the future DAG looks like. For instance, consider the DAG shown in figure 6.2b. Again, critical sections in each layer access the same lock. However, now there is no synchronization point between layers; therefore, the layers can execute in parallel. An optimal offline scheduler will schedule the right-most critical section of each layer first so it can schedule the next layer in parallel with the first layer. Therefore, if we carefully construct the DAG, an offline scheduler can get full speedup. However, an online scheduler can not know which critical section of each layer leads to more future work. Therefore, it may make the mistake of executing the right-most critical section last for each layer and therefore will get no speedup. In particular, for any online strategy S, there is a bad DAG where the next layer is always created by the critical section this strategy S executes last. Therefore, an online scheduler can not guarantee optimality.

Let us now consider replay. In this case, T_1/P, and T'_∞ are lower bounds; therefore the replay bound of $O(T_1/P + \lg\lg P T'_\infty)$ is nearly tight — it just has an additional $\lg\lg P$ factor on the span term which is tiny for most machines. In addition, since it is on the span term, according to the work-first principle [77], this overhead does not affect computations with sufficient parallelism.

On series-parallel (or more generally, fully-strict) computations, depth-first work stealing (of the kind we use) also guarantees a space bound; in particular, if the sequential execution uses S_1 stack space, work-stealing uses $O(PS_1)$ when using P workers. Since record uses vanilla work-stealing, it also provides this space bound. However, the replay scheduler executes the augmented DAG which is not a fully-strict DAG. In fact, one can generate augmented DAGs for which it would be impossible to simultaneously provide good speedup and space bounds;

the construction is similar to the lower bound in section 3.1 in [28]. Since our replay scheduler provides good speedup, it can not guarantee low space usage.

6.3 Empirical Evaluation

We now turn to the implementation and evaluation of PORRidge. The main benefit of a processor-oblivious record-replay system is that one can replay an execution on a different number of processors from that used during the recording — including a larger number — allowing the replay to benefit from parallel execution. There are inherent overheads in the record and replay in order to allow processor-oblivious replay, however. Specifically, during record, PORRidge must record happens-before edges via strand IDs in a schedule-independent fashion; during replay, PORRidge may need to suspend and resume strands upon lock acquires and releases.

We empirically evaluated the overhead and scalability of the record phase and replay phase across six benchmarks with different execution characteristics. Our results indicate that, as long as the computation has enough parallelism, the record phase scales similarly as the baseline. When the baseline runs low on parallelism, however, the record phase can incur high overhead and cause slow down. The replay phase tends to incur similar and sometimes smaller overhead than the recording. Due to its non-blocking execution model, replay using the same number of workers as in the recorded execution tends to execute faster than recording. In fact, as long as the recorded execution contains sufficient parallelism, the replay continues to get speedup beyond P workers, where P is the number of workers used during recording.

Benchmarks. We used the following six benchmarks to evaluate the PORRidge system. The first one, `chess`, is a Cilk Plus program published by Intel [101] that solves a chess

	number	*number of lock acquires*				
application	*of locks*	*total*	*min*	*max*	*mean*	*std. dev.*
`chess`	4	2.8e4	0	2.8e4	7.1e3	1.4e4
`dedup`	1	7.3e5	7.3e5	7.3e5	7.3e5	n/a
`ferret`	1	256	256	256	256	n/a
`matching`	5e6	5e7	5	25	10	2.23
`MIS`	5e6	2.8e6	3	27	5.63	2.73
`refine`	4.8e7	1.2e7	0	27	0.26	0.56

Table 6.1: Application benchmarks used and their execution characteristics measured when running on one worker. The total column shows the total number of lock acquires across all locks during execution. The min column shows the minimum number of lock acquires invoked on a given lock across all locks; similarly, the max column shows the maximum. The last two columns show the average number of lock acquires per lock and the standard deviation.

puzzle — given eight chess pieces excluding pawns, count the number of configurations where the pieces are placed such that they can attack all squares on an 8×8 chess board. The program uses reducers [79] to keep counts on the number of such configurations found and to perform I/O; we modified the program to use locks instead. Two benchmarks, `dedup` and `ferret`, from the PARSEC benchmark suite [26, 25] are used; they can be implemented as Cilk Plus programs that utilize reducers for performing file I/O, which we replace with locks. Finally, we converted three nondeterministic versions of graph algorithms from the Problem Based Benchmark Suite [169] to use locks instead of Compare-And-Swap (CAS): `MIS` (Maximal Independent Set), `matching` (Maximal Matching), and `refine` (Delaunay Refinement). These benchmarks cover a wide spectrum of behaviors.

Their runtime characteristics when executing on one worker are shown in table 6.1. Note that the characteristics during parallel execution may differ slightly for some of the graph benchmarks as they are nondeterministic by nature. The first three benchmarks use few locks, but still have plenty of lock acquires; however, they do a significant amount of work outside of critical sections. The three graph benchmarks use a much larger number of locks,

since there is one lock per vertex. In addition, they do almost all of their work within critical sections.

Experimental Platform. We ran our experiments on an Intel Xeon E5-2665 with 16 2.40-GHz cores on two sockets; 64 GB of DRAM; two 20 MB L3 caches, each shared among 8 cores; and private L2- and L1-caches of sizes 2 MB and 512 KB, respectively. Both hyperthreading and dynamic frequency scaling are disabled in order to get consistent results across runs. For recorded runs, running times are in seconds as a mean of five runs. For a given number of workers, the one with the median running time is chosen to be used for the replay runs. For replay runs, running times are in seconds as a mean of ten runs. For the most part, the standard deviation was within 5% of the mean for both record and replay. There are a few exceptions: graph algorithms have higher standard deviation during replay. In particular, memory-bound graph algorithms (`matching` and `MIS`) have higher standard deviations during some replayed runs, up to 12% for `MIS`. for this phenomenon in the next section.

Notation. We use the following notations in this section. The label **baseline** refers to executions of the benchmarks with ordinary spin locks (i.e., without PORRidge). The label **record** refers to the executions with recording enabled using PORRidge. The label **replay** refers to the executions with replay enabled using PORRidge. We use P_{base} to refer to the number of workers used during baseline execution, P_{rec} to refer to the number of workers used during record and P_{rep} to refer to the number of workers used during replay. When P_{rep} is subscripted, the subscript refers to the number of workers used in the recorded that is being replayed; if there is no subscript, then we are replaying with the same number of workers as the recorded run.

	baseline	*record*	*replay*
`chess`	64.43	64.38 (1.00×)	65.11 (1.01×)
`dedup`	48.04	48.20 (1.00×)	48.16 (1.00×)
`ferret`	8.92	8.89 (1.00×)	9.10 (1.02×)
`matching`	3.06	9.64 (3.15×)	10.07 (3.29×)
`MIS`	1.01	3.42 (3.39×)	3.77 (3.73×)
`refine`	11.70	14.73 (1.26×)	13.63 (1.16×)

Table 6.2: Execution times running on one worker ($P_{base} = P_{rec} = P_{rep} = 1$) for six benchmarks, in seconds. The `replay` column shows the replay execution time for replaying the run recorded with one worker. The numbers shown in parenthesis indicate the overhead compared to the baseline.

6.3.1 Overhead of Record

To evaluate the recording overhead, we compare the running time of PORRidge recording on one worker with the baseline running on one worker. table 6.2 shows the execution times of six benchmark for these configurations. The recording overhead ranges from 1–3.39× with a geometric mean of 1.62×. Since PORRidge incurs overhead only upon lock operations, the overhead is in part dictated by how much work is done per lock acquire. For programs that perform sufficient amount of work outside of critical sections, such as `chess`, `dedup`, and `ferret`, the overhead is negligible. The graph algorithms, especially `matching` and `MIS`, incur higher overhead. For these applications, almost all of the work occurs inside critical sections. In addition, each critical section does a very small amount of work. Their executions mostly involve repeatedly traversing some edge, acquiring a lock corresponding to the vertex at the end of the edge, updating a field in the vertex, and releasing the lock. Hence, the execution time of these programs is dominated by the cost of acquiring and releasing locks. Moreover, these applications are memory bound — they have large working sets and display very little locality in accessing data. The additional space used for logging during recording puts additional pressure on the memory hierarchy. In the initial implementation, we have used a hash list to detect collisions of section IDs (discussed in section 6.1), and the additional cache

	replay on one, recorded on P					
application	*P = 1*	*P = 2*	*P = 4*	*P = 8*	*P = 12*	*P = 16*
`chess`	65.14 (1.01×)	65.13 (1.01×)	65.11 (1.01×)	65.17 (1.01×)	65.20 (1.01×)	65.13 (1.01×)
`dedup`	48.16 (1.00×)	48.15 (1.00×)	48.16 (1.00×)	48.21 (1.00×)	48.11 (1.00×)	48.20 (1.00×)
`ferret`	9.10 (1.02×)	8.95 (1.01×)	8.95 (1.01×)	8.94 (1.01×)	8.93 (1.00×)	8.93 (1.00×)
`matching`	10.07 (1.04×)	10.13 (1.05×)	10.13 (1.05×)	10.17 (1.05×)	10.37 (1.08×)	10.43 (1.08×)
`MIS`	3.77 (1.11×)	3.90 (1.15×)	3.94 (1.16×)	3.93 (1.16×)	3.97 (1.17×)	4.04 (1.19×)
`refine`	13.63 (0.93×)	13.67 (0.93×)	13.47 (0.91×)	13.73 (0.93×)	13.70 (0.93×)	13.73 (0.93×)

Table 6.3: Execution times, in seconds, when replaying on one worker executions recorded on different number of workers. The numbers shown in parenthesis indicate the overhead compared to the execution time of that recorded on one worker. Highlighted cells indicate execution time differ from that of the recorded run on one worker by more than ±10%.

misses incurred by accessing the hash list incurred much larger overhead in these applications (8–9×). By reordering the bookkeeping data layout to obtain better spatial locality and using a bloom filter instead of a hash list, we were able to reduce the overhead drastically.

6.3.2 Overhead of Replay

Replay has two types of overheads. Replay, like record, incurs overhead upon lock acquires and releases. When a worker tries to acquire a lock, it must access that lock's hash-list and query the hash-list with the current strand ID to see if this strand is the next in line to access this lock. If so, it can acquire the lock. Otherwise, it must suspend. Upon a release, the worker advances the head of the hash-list; if the next lock acquire has been tried and suspended, the worker suspends its current execution and resumes the execution of the next lock acquire. In addition, replay also incurs overheads due to maintaining more deques than the vanilla Cilk runtime system.

If we record on one worker and replay on one worker, the execution proceeds in exactly the same order. Therefore, the replay execution never has to suspend. Essentially, the work done by replay is the same as the work done by record except that replay reads the hash-list

instead of writing to it. We see from table 6.3 that in this case, as expected, replay exhibits essentially the same overhead as record.

The more interesting case is when we record on more than one worker and then replay on one worker ($P_{rec} > 1$ and $P_{rep} = 1$). In this case, replay has additional overheads — namely the overhead of suspending and resuming lock acquires. Table 6.3 shows the overhead of replay on one worker, replaying executions recorded on different number of workers. We can compare these with the execution when we record one worker and replay on one worker (the first column of the figure). It turns out that for the most part, the additional overhead incurred by processor-oblivious replay is small. Only `MIS` incurs more than 10% additional overhead compared to one-worker record. Note that when we replay (on one worker) an execution recorded on multiple workers, the worker likely encounters critical sections in a different order than the recorded execution did. When this worker encounters a critical section that cannot be executed yet it must suspend its current deque and work steal. In addition, since work stealing is random, the next critical section it acquires may again not be the right one. Therefore, the worker may suspend many deques before encountering a critical section it can execute. Since these graph benchmarks have very little work outside critical sections and a very large number of lock acquires, these suspensions (and the cost of resuming later) cause these overheads.

We can gauge such an overhead by comparing the overhead of executions with $P_{rec} > 1$ and $P_{rep} = 1$ with the overhead of executions with $P_{rec} = P_{rep} = 1$ shown in table 6.3. It turns out that for the most part, the additional overhead incurred by suspending and resuming lock acquires is negligible — at most a few percent increase for the memory-bound benchmarks.

6.3.3 Scalability

To analyze the scalability of PORRidge for recording, we compare the speedup of record to the baseline's. The speedup is computed with respect to their respective one-worker execution counterpart. Table 6.4 shows scalability of both the baseline and recorded runs across benchmarks (the first two columns). The scalability profile for record tracks that of the baseline closely across all benchmarks. This is especially surprising for memory-bound benchmarks since the workers spend longer within critical sections during recording compared to the baseline. In spite if this, it appears that the additional overhead is distributed across processors evenly and did not reduce the overall parallelism by much.

Table 6.4 as well shows scalability of replay runs that replay executions recorded on $P_{rec} = 1, 2, 4, 8, 12, 16$ processors. Here, we measure the speedup of a replay run by comparing it against the time replaying the same recorded execution on one worker.

Recall that the expected execution time for replay on P workers is $O\left(T_1/P' + T'_\infty \log\log P'\right)$, where T_1 is the overall work in the computation and the T'_∞ is the span in the augmented dag. Since $T_\infty \leq T'_\infty \leq T_\infty + B$, replay should scale as long as record scales (ignoring the $\lg\lg P$ term). The experiments do indicate that it is generally safe to ignore the $\lg\lg P$ term and that the overheads of suspending and restarting in replay is small. For applications where the recording scales, indeed we see that the replay on the same number of workers scale similarly, as shown in the highlighted cells in table 6.4.

The two exceptions are data points in `matching` and `MIS`. There are two possible explanation. The first is that the augmented dag is running out parallelism. We don't believe that this is the case, since if replay uses more workers $P_{rep} > P_{rec}$, we continue to see the execution scales (i.e., by looking at the scalability of data points below the highlighted cells). The more likely

explanation is the following: these benchmarks are already memory bound, and replay has a much larger memory footprint than record, causing additional cache misses, and the higher memory latency slows down the parallel execution. Indeed, these executions incur higher cache misses during replay than during record. There are two reasons for these additional cache misses. First, during replay, workers suspend their current deque from time to time (discussed in section 6.1) and thus can create large number of suspended deques. Second, while record can use a bloom filter and do without a hash list, replay must use either an array or a hash list. While arrays have fewer cache misses than the hash list, both of these have a larger memory footprint than the bloom filter. Indeed, recall that one of the optimizations that we implemented for replay is to use a lock-acquire array instead of a hash list if the number of lock acquires per lock is small (which is the case for these benchmarks). The overhead of replay was much higher when we used a hash list in our initial implementation, which requires even more memory than the array.

The additional memory footprint also in part explains the higher standard deviation for some benchmarks mentioned earlier, namely `matching` and `MIS`. How many additional deques created during replay is a function of scheduling, and the number of suspended deques can differ from run to run. Execution times for benchmarks that are already memory bound will be more sensitive to this changes in the number of suspended deques.

Finally, note that, replay of an execution recorded on $P_{rec} = P$ workers can continue to scale when $P_{rep} > P$, as long as there is still parallelism in the augmented dag, as predicted by our theoretical analysis. Such scalability is evident across all benchmarks, as shown by the numbers in the same column below the highlighted cells in table 6.4.

bench	P	*baseline*	*record*	*replay on P workers ($P_{rep} = P$) an execution recorded on P' workers ($P_{rec} = P'$)*					
				P' = 1	*P' = 2*	*P' = 4*	*P' = 8*	*P' = 12*	*P' = 16*
chess	1	64.49 (1.00×)	64.36 (1.00×)	65.14 (1.00×)	65.13 (1.00×)	65.11 (1.00×)	65.17 (1.00×)	65.20 (1.00×)	65.13 (1.00×)
	2	32.20 (2.00×)	32.20 (2.00×)	32.60 (2.00×)	32.60 (2.00×)	32.61 (2.00×)	32.64 (2.00×)	32.63 (2.00×)	32.66 (1.99×)
	4	16.11 (4.00×)	16.11 (4.00×)	16.50 (3.95×)	16.36 (3.98×)	16.35 (3.98×)	16.37 (3.98×)	16.35 (3.99×)	16.34 (3.99×)
	8	8.15 (7.91×)	8.13 (7.92×)	8.55 (7.62×)	8.31 (7.84×)	8.42 (7.73×)	8.45 (7.71×)	8.34 (7.82×)	8.31 (7.84×)
	12	5.38 (11.99×)	5.38 (11.96×)	5.91 (11.02×)	5.65 (11.53×)	5.75 (11.32×)	5.75 (11.33×)	5.66 (11.52×)	5.63 (11.57×)
	16	4.04 (15.96×)	4.11 (15.66×)	4.52 (14.41×)	4.50 (14.47×)	4.47 (14.57×)	4.57 (14.26×)	4.32 (15.09×)	4.35 (14.97×)
dedup	1	48.04 (1.00×)	48.20 (1.00×)	48.16 (1.00×)	49.15 (1.00×)	48.16 (1.00×)	48.21 (1.00×)	48.11 (1.00×)	48.20 (1.00×)
	2	24.43 (1.87×)	24.44 (1.94×)	24.58 (1.92×)	24.44 (1.88×)	24.44 (1.95×)	24.43 (1.95×)	24.45 (1.96×)	24.42 (1.94×)
	4	12.43 (3.86×)	12.40 (3.89×)	12.61 (3.82×)	12.55 (3.84×)	12.51 (3.85×)	12.40 (3.89×)	12.40 (3.88×)	12.41 (3.88×)
	8	6.43 (7.47×)	6.49 (7.43×)	6.72 (7.17×)	6.69 (7.20×)	6.61 (7.29×)	6.46 (7.47×)	6.45 (7.46×)	6.46 (7.46×)
	12	4.52 (10.63×)	4.53 (10.64×)	4.88 (9.87×)	4.83 (9.97×)	4.75 (10.14×)	4.63 (10.41×)	4.55 (10.57×)	4.55 (10.59×)
	16	3.61 (13.31×)	3.65 (13.32×)	3.95 (12.19×)	3.94 (12.22×)	3.89 (12.38×)	3.76 (12.82×)	3.69 (13.04×)	3.64 (13.24×)
ferret	1	8.92 (1.00×)	8.89 (1.00×)	9.10 (1.00×)	8.95 (1.00×)	8.95 (1.00×)	8.94 (1.00×)	8.93 (1.00×)	8.93 (1.00×)
	2	4.52 (1.97×)	4.57 (1.95×)	4.53 (2.01×)	4.54 (1.97×)	4.56 (1.96×)	4.52 (1.98×)	4.53 (1.97×)	4.52 (1.98×)
	4	2.31 (3.86×)	2.33 (3.82×)	2.32 (3.92×)	2.34 (3.82×)	2.32 (3.86×)	2.32 (3.85×)	2.31 (3.87×)	2.32 (3.85×)
	8	1.27 (7.02×)	1.24 (7.17×)	1.24 (7.34×)	1.24 (7.22×)	1.27 (7.05×)	1.26 (7.10×)	1.26 (7.09×)	1.26 (7.09×)
	12	0.91 (9.80×)	0.91 (9.77×)	0.93 (9.78×)	0.90 (9.94×)	0.91 (9.84×)	0.92 (9.72×)	0.92 (9.71×)	0.91 (9.81×)
	16	0.76 (11.74×)	0.78 (11.40×)	0.75 (12.13×)	0.75 (11.93×)	0.75 (11.93×)	0.75 (11.92×)	0.75 (11.91×)	0.74 (12.07×)
matching	1	3.06 (1.00×)	9.64 (1.00×)	10.07 (1.00×)	10.13 (1.00×)	10.13 (1.00×)	10.17 (1.00×)	10.37 (1.00×)	10.43 (1.00×)
	2	1.92 (1.67×)	5.78 (1.46×)	6.88 (1.49×)	6.82 (1.49×)	6.93 (1.46×)	6.64 (1.53×)	6.68 (1.55×)	6.63 (1.57×)
	4	0.96 (3.19×)	3.03 (3.18×)	3.95 (2.55×)	3.90 (2.60×)	3.75 (2.70×)	3.64 (2.79×)	3.63 (2.86×)	3.62 (2.88×)
	8	0.50 (6.12×)	1.68 (5.74×)	3.77 (4.31×)	3.99 (4.57×)	4.22 (4.61×)	2.22 (4.58×)	2.21 (4.69×)	2.11 (4.94×)
	12	0.32 (9.56×)	1.13 (8.53×)	2.57 (3.92×)	2.42 (4.19×)	2.21 (4.58×)	1.89 (5.38×)	1.77 (5.86×)	1.68 (6.21×)
	16	0.25 (12.24×)	0.89 (10.83×)	2.56 (3.93×)	2.41 (4.20×)	2.11 (4.80×)	1.79 (5.68×)	1.58 (6.56×)	1.57 (6.64×)
MIS	1	1.02 (1.00×)	3.40 (1.00×)	3.77 (1.00×)	3.90 (1.00×)	3.94 (1.00×)	3.93 (1.00×)	3.97 (1.00×)	4.04 (1.00×)
	2	0.65 (1.57×)	2.03 (1.67×)	2.57 (1.47×)	2.54 (1.54×)	2.48 (1.59×)	2.47 (1.59×)	2.45 (1.62×)	2.52 (1.60×)
	4	0.32 (3.19×)	1.03 (3.30×)	1.63 (2.31×)	1.52 (2.57×)	1.36 (2.90×)	1.34 (2.93×)	1.33 (2.98×)	1.32 (3.06×)
	8	0.16 (6.38×)	0.58 (5.86×)	1.38 (2.73×)	1.14 (3.42×)	0.93 (4.24×)	0.79 (4.97×)	0.81 (4.90×)	0.79 (5.11×)
	12	0.13 (7.85×)	0.22 (8.10×)	1.54 (2.45×)	1.26 (3.10×)	0.89 (4.43×)	0.69 (5.70×)	0.61 (6.51×)	0.67 (6.03×)
	16	0.14 (7.29×)	0.38 (8.95×)	1.50 (2.51×)	1.28 (3.05×)	0.94 (4.19×)	0.73 (5.38×)	0.61 (6.51×)	0.63 (6.41×)
refine	1	11.70 (1.00×)	14.73 (1.00×)	13.63 (1.00×)	13.67 (1.00×)	13.47 (1.00×)	13.73 (1.00×)	13.70 (1.00×)	13.73 (1.00×)
	2	7.36 (1.59×)	9.32 (1.58×)	9.38 (1.45×)	9.11 (1.50×)	9.19 (1.47×)	9.32 (1.47×)	9.13 (1.50×)	9.17 (1.50×)
	4	4.40 (2.66×)	5.53 (2.66×)	5.60 (2.43×)	5.49 (2.49×)	5.35 (2.52×)	5.36 (2.56×)	5.26 (2.60×)	5.34 (2.57×)
	8	3.15 (3.71×)	3.87 (3.81×)	4.29 (3.18×)	3.99 (3.43×)	3.90 (3.45×)	3.77 (3.64×)	3.74 (3.66×)	3.66 (3.75×)
	12	2.73 (4.29×)	3.32 (4.44×)	3.91 (3.49×)	3.62 (3.78×)	3.44 (3.92×)	3.36 (4.09×)	3.36 (4.08×)	3.31 (4.15×)
	16	2.45 (4.78×)	3.04 (4.86×)	3.61 (3.78×)	3.39 (4.03×)	3.21 (4.20×)	3.04 (4.52×)	3.00 (4.57×)	2.96 (4.64×)

Table 6.4: Execution times on $P = 1, 2, 4, 8, 12, 16$, in seconds, and their scalability profile for all benchmarks. Each of the replay columns shows the replay time with $P_{rep} = P$ workers replaying the same recorded execution (with $P_{rec} = P'$, as shown in the column heading). The numbers in the parenthesis indicate the speedup comparing to its single-worker execution counterpart, which has the 1.00 speedup. The highlighted cells indicate replay runs that uses the same number of workers as in the recording.

Benefits of processor obliviousness. As our experimental data indicates, processor-oblivious record and replay can be implemented efficiently. The only time PORRidge exhibits non-negligible overhead is when the benchmark is already memory bound. For dynamic multithreaded computations, a processor-aware record-and-replay system would need to log additional information to record the inherent non-determinism in the scheduler, which would further increase the memory footprint of the recording.

Moreover, the strategy used by PORRidge has the additional benefit of scaling the replay beyond the number workers used in record, which a processor-aware record and replay system cannot provide. If a recording is done on P workers and takes x time, in a processor-aware system, the replay cannot run in less than x time (asymptotically) no matter how many workers we give it. In fact, it would likely be slower since it would spin when it encounters a lock acquire that is not ready and would have to exactly replicate the steal patterns of the record, causing potentially more idleness. On the other hand, PORRidge never spins during replay and uses suspension to explore all the possible parallelism in the augmented dag. Therefore, as the experiments indicate, replay often runs just as fast as the record when $P_{rep} = P_{rec}$, and can continue to scale when $P_{rep} > P_{rec}$.

6.4 Related Work

Record and Replay. To our knowledge, all software-based record and replay systems are tied to thread-based programming models: a runtime system records the behavior and interleaving of the threads in the program, and on replay re-runs the same threads with the same behavior. Recording and replaying on the same number of threads simplifies both the recording process (as thread-based identifiers can be used to identify operations) and the replay process (as there is no need to map operations from the recorded run onto a different

number of threads). RecPlay [157] and JaRec [82] do not handle racy accesses, and have reasonable overhead, but, as with PORRidge, are unsound in the presence of races.

Racy accesses are more challenging, since accesses to shared memory result in happens-before edges that must be preserved during replay. For systems that handle racy accesses, there are several approaches. Some speculate that races are infrequent or irrelevant to keep recording overhead down [118, 186]. Some preserve a limited amount of information during record and rely on offline search or constraint-solving approaches to generate the information required for replay [125, 6, 97, 146]. Some systems track racy interleavings directly, which either add large overhead [190, 116], use coarse-granularity communication tracking (such as page-based conflict detection) that can be overly-conservative [113, 63], or rely on carefully modified virtual machines [33].

One could apply a traditional thread-based record-and-replay system on dynamic multi-threaded computation directly, and record all sources of non-determinism in order to replay deterministically. PinPlay [147] is such a general record and replay system based on Pin, a popular dynamic binary instrumentation framework [128], that captures all sources of non-determinism including racy memory accesses, thread interleavings, and results from system calls. We ran PinPlay on Delaunay Refinement (`refine` described in section 6.3) and find that it has 96.8× overhead for recording and 16.1× overhead for replay when executing the computation on one worker — 1-2 orders of magnitudes worse than the PORRidge overheads of 1.26× and 1.16×, respectively. When we tried recording and replaying on multiple workers, the executions with PinPlay slowed down and showed no speedup. This result in part speaks to the performance advantage of PORRidge's approach, because PORRidge does not need to reproduce the runtime's non-determinism while traditional thread-based record-and-replay systems must. Reproducing the runtime's non-determinism requires logging all inter-thread

interactions among worker threads and causing a worker thread to spin wait (instead of doing useful work) when it reaches a recorded inter-thread interaction before the other thread gets there. Note that such inter-thread interactions include all failed steal attempts between a thief and a victim worker, since a failed steal attempt is communicated through shared memory accesses. Chimera [117], another record-and-replay system for pthreaded code, on the other hand, uses static race detection to identify potentially-racing pairs of accesses, and uses lightweight synchronization, as well as lock coarsening, to enable a simple record and replay technique. Such an approach, could be adapted to make PORRidge applicable to racy Cilk programs.

Another strategy is to record information at the *hardware* level [189, 96, 136, 149], by piggybacking on cache-coherence protocols to record communication between different hardware contexts. While these systems could, in principle, be used to record the behavior of Cilk programs and to capture the non-determinism introduced by the scheduler, they have two drawbacks: 1) like existing software-based models, their (hardware) context-based recording system constrains replay to run with the same level of parallelism as record; 2) they require hardware modifications, and hence do not work in any existing commodity systems.

Determinism. A related technique is *deterministic execution*, where a combination of programming model constraints and runtime checks ensures that an application always produces the same behavior when presented with the same input. Note that this is subtly different than record and replay: in record and replay, different *recorded* runs can exhibit different behaviors; replay must replicate whichever recorded run it is replaying. One approach to determinism is to mandate it through programming model restrictions [27, 39, 139, 156, 32, 24] which generally preclude general use of locks and other synchronization mechanisms.

Moreover, while some of these approaches can provide determinism independent of the number of threads [27, 139], most do not. Another approach is to enforce determinism through hardware [54, 56], compiler [22], OS [10, 23] or runtime approaches [126, 142]. While these techniques do not require specialized programming models, these techniques are usually not processor oblivious.

6.5 Conclusions and Future Work

This chapter presented the first processor oblivious record and replay scheme for data race-free dynamic multithreaded programs. This scheme is provably good, efficient in practice, and provides good scalability.

There are many directions of future work. First, we could target a richer set of primitives that induce happens-before relationships; for instance, `try-lock` and `compare-and-swap`. These require rethinking the exact semantics we want from a happens-before edge, since, in some cases, programs use the non-determinism induced by these mechanisms to enable efficiency, complicating which edges we want to record. Second, we could try to expand to programs with data races — this would involve recording happens-before relationships not just between critical sections, but also between accesses to memory locations that could be involved in races. As mentioned in the introduction, this does not require conceptual changes to PORRidge, just the ability to indicate to PORRidge where non-determinism due to races might occur. Finally, we can explore other mechanisms to enable processor-oblivious record and replay to see if some of them will give better performance.

Chapter 7

Efficient Race Detection for Structured Future-Parallel Computations

The race detection algorithm in chapter 4 was focused only on fork/join computations. These fork/join computations can be represented as series-parallel DAGs, yielding nice structural properties that allowed CRacer to efficiently detect determinacy races. Specifically, a reachability data structure can be constructed on-the-fly and queried with no asymptotic overhead, whether execution is serial [19] or in parallel (chapter 4).

This chapter and the next consider programs with more general dependencies. In particular, we consider the use of futures, as described in section 5.2.1. Notice that two primitives for futures (`create_future` and `get_future`) subsume the `spawn` and `sync` primitives for fork/join parallelism. The `spawn` primitive is a restricted `create_future`, since corresponding subcomputations cannot escape the scope of the subsequent `sync` primitive. In fact, using futures can generate arbitrary dependencies. This is beneficial since not all parallel algorithms can be written as fork/join computations.

The disadvantage, however, is that non-series-parallel computation DAGs are harder to reason about. Hence the ability to detect determinacy races is even more important than for series-parallel programs.

This chapter presents a race detection algorithm for programs that use futures in a "structured" way (see chapter 8 for an approach to arbitrary futures). Specifically, we only consider programs with **structured futures**, which must be used in the following way:

- `get_future` is called on each future handle at most once
- there is a sequential dependency chain from each `create_future` to the corresponding `get_future`

Intuitively, the second restriction ensures that there can be no races on future handles. These restrictions are equivalent to single-touch well-structured futures analyzed by Herlihy and Liu [92]; these computation can be executed by a work-stealing scheduler efficiently and with relatively few cache misses (compared to arbitrary future-parallel computations). Such restrictions could be enforced by a compiler by ensuring that future handles are only passed by value (as parameters to functions or as return values), never captured as a reference, and cannot be written to a global variable that can be read by multiple parallel subcomputations.

We first describe an algorithm for maintaining reachability relationships among strands in a computation. Pairing our reachability algorithm with an access history data structure (see chapter 4) yields a race detection algorithm that runs a program serially and in time $O(T_1\alpha(m,n))$, where T_1 is the work of the program, α is the inverse Ackermann's function, m is the number of memory accesses in the program, and n is the number of parallel strands created (the number of `spawn` or `get_future` operations).

Section 7.1 discusses how to maintain reachability for programs with structured use of futures by extending the SP-Bags algorithm; section 7.2 proves its correctness. The necessary changes to a series-parallel access history data structure are discussed in section 7.3, which also analyzes the runtime of the full race detection algorithm. We briefly discuss related work in section 7.4 before concluding in section 7.5.

7.1 Extending SP-Bags for Structured Futures

As mentioned in the introduction, programmatically, `spawn` and `sync` are subsumed by `create_future` and `get_future` since we can model `spawn` as `create_future` and `sync` as `get_future` on all the functions in the function scope. Henceforth, for simplicity, we will assume that we only have `create_future` and `get_future` constructs to create parallelism.

We refer to the strand that calls `get_future (F)` (as its last instruction) as the `creator(F)` and strand that calls `get_future (F)` (as its first instruction) as the `getter(F)`. The computation dag consists of three kinds of nodes — regular strands with one incoming and one outgoing edge, creator strands with two outgoing edges and getter strands with two incoming edges. It also consists of three kinds of edges: **spawn** edges are edges from creator nodes to the first strand of the future; **join** edges are edges from last strand of a future to getter nodes; all other edges (that go between strands of the same function instance) are **continue** edges. [1]

This algorithm is similar to the SP-Bags algorithm for detecting races for series-parallel programs [71]. As with that algorithm, we will use a the fast disjoint-set data structure [179].

[1] Spawn and join edges here were called non-SP edges earlier, but it is more convenient to distinguish between them in this manner in this section.

The data structure maintains a dynamic collection D of disjoint sets and provides three operations:

- MAKE-SET$(x) : D = D \cup \{\{x\}\}$
- UNION$(A, B) : D = D \cup A \cup B - A - B$
- FIND(x) returns the set that contains the element x.

In addition, like SP-Bags, our algorithm depends on the **depth-first eager execution** of the computation. That is, when we get to a `creator(F)`, we will always execute the future `F`. When we return from the future `F`, we will then follow the continue edge of `creator(F)`. Due to the restriction that we have structured futures, this execution has the property that the execution will never block since when we get to the `getter(F)` strand, we will have already executed `F` (which executes immediately after `creator(F)`.

Now we are ready to describe the algorithm. This algorithm maintains a bag for each function instance `F` for which `get_future` has not yet been called (these bags can be stored with the future handle). This bag is labeled either an S-bag, represented by S_F or a P-bag, represented by P_F. The algorithm maintains these bags as follows:

1. On $f =$ `create_future`$(F) : S_F =$ MAKE-SET(u) where u is the first strand of F.
2. On return of function F: $P_F = S_F$; de-allocate S_F.
3. On $y =$ `get_future`(f) where f is F's handle and the current strand (`getter`(F)) is in function F': $S_{F'} =$ UNION$(S_{F'}, P_F)$; de-allocate P_F.
4. strands of a function F are always added to S_F.

In order to check if the currently executing strand v (which reads or writes a memory location ℓ) is in parallel with a (already executed) strand u (from ℓ's access history), we do a FIND(u). If this returns an S bag, then $u \prec v$ and u's access doesn't race with v's; otherwise it does.

7.2 Correctness Proof

We will prove the following theorem:

Theorem 7.1. *If the currently executing strand is v, then a previously executed strand u is currently in an S bag iff $u \prec v$; otherwise it is in a P bag.*

We now define a couple of more terms. A node u is a **spawn predecessor** of a node v if there is a path from u to v which consists of only spawn and continue edges. A node u is a **join predecessor** of v if there is a path from u to v that consists of only join and continue edges. Spawn and join-successor are defined in the symmetric way. Each node is its own spawn and join predecessor and successor.

We first state a property of eager executions in this model:

Property 7.2. When a strand v is currently executing, all spawn predecessors of v are part of some active function — a function that has started, but not completed. In addition, the converse is also true; all strands w that are part of active functions are spawn predecessors of v.

The proof consists of two steps: (1) We will first establish a static property of paths in programs with structured futures; in particular, if $u \prec v$, then we can find a node w where the path from u to w where u is a join-predecessor of w and w is a spawn predecessor of v. (2) We will then establish a dynamic property of the algorithm, namely that if u is a join

predecessor of some node w, where w is part of an active function G, then u is in the S bag of G. If none of u's join successors are part of an active function, then u is in a P bag. Together, these statements along with property 7.2 prove theorem 7.1.

First, we define a canonical order on the futures of the program. The order starts with the main function; we then progressively add futures to the computation. A future F can only be added if its creator strand and getter strand have already both been added. We can show that we can always find a canonical order by induction.

Lemma 7.3. *We can always find a canonical order.*

Proof. We will induct on adding new futures. We can always start since the main function is added first. At some point, we have already added some futures, say a set S. Next, we will add the future F such that there is no future G such that $\texttt{creator}(F) \prec \texttt{creator}(G)$ where both creator nodes are in S. There must be some such future (if there are more than one, we can choose arbitrarily). Since no future was created sequentially after F was created, and $\texttt{getter}(F)$ is sequentially after $\texttt{creator}(F)$, $\texttt{getter}(F)$ must also be in S; therefore, it is legal for us to pick F as the next future in our canonical order. □

We can now show this property by inducting on futures in the canonical order. It is easy to see that as we add futures, we do not change reachability between nodes that already exists and for new nodes, we augment additional paths by adding spawn edges to the end of the path and join edges to the beginning. The next lemmas follow directly.

Lemma 7.4. *If $u \prec v$, then there exists a path from u to v that contains two sections: The first path (possibly empty) contains only join and continue edges and the second part (possibly empty) contains only spawn and continue edges. In other words there is never a spawn edge followed by a join edge on this path. In addition, this path is unique.*

Proof. Induct on futures in the canonical order (which we can always find according to lemma 7.3) and show that this is true as we add futures one by one.

Base case: We first have only the main strand, so this is true trivially.

Inductive case: Assume that after we have added a set S of futures, the statement is true. We now add a new future F'.

Consider any nodes u and v in this new dag where . If neither u nor v are in F', then the addition of F' does not add any new paths between u and v (since the only new path added is between $\texttt{creator}(F)$ and $\texttt{getter}(F)$ and there was already a path between them before we added F'). In addition, any new path added does have a spawn followed by a join — therefore, the uniqueness is preserved. Therefore, we only need consider pairs where either u or v are in F'. If u is in F' and v is not, then the path from u to v must go from the last strand of F' to $\texttt{getter}(F')$ and then to v. By inductive hypothesis, the path from $\texttt{getter}(F')$ already follows the desired property and the path from u to $\texttt{getter}(F')$ only contains join and continue edges. Therefore, the property still holds. A symmetric argument applies when v is in F'. □

Lemma 7.5. *If $u \prec v$, then there is some node w (possibly u or v) which is a join successor of u and a spawn predecessor of v.*

We next prove the dynamic property by looking at the execution as it unfolds. Therefore, we implicitly assume that when we refer to any strand (or function), it is either currently executing or has already executed (we have no knowledge of strands or functions that are still to execute).

For each function F, we define its **operating function** G as the function containing the "farthest join descendant" of the last executed strand of F. We say that a function F is **active** if it has started, but has not yet returned. It is its own operating function. If a function is not active (it has returned), it may be **confluent** or **non-confluent**. It is confluent if its operating function is active; otherwise it is non-confluent. By definition, the operating function of a non-confluent function is always some non-confluent function and an operating function is always either active or non-confluent. A confluent function can never be its own operating function, but a non confluent function F may be its own operating function if its `getter`(F) has not yet executed.

The following lemma is proved by induction on the program as it executes.

Lemma 7.6. *(a) When a function F is active, all its strands are in its S bag. (b) If a function F is confluent, then all its strands are in its operating function G's S bag. (c) If a function F is non confluent, then all its strands are in its operating function G's P bag.*

Proof. When a function is first called, it has an S bag and its strands are placed in the S bag. They remain in this S bag while it is active. Once the function returns, all its items move to a P bag All its strands remain in S bags while the function is active. For the other two statements, we induct on time after F returns.

Base Case: When F returns, `getter`(F) has not yet been called. Therefore, it is its own operating function; it is non confluent; and all its strands are in its own P bag.

Inductive Case: We will do this by two cases:

Case 1: F is non-confluent and G is its (non-confluent) operating function; by inductive hypothesis, all strands of G are in G's P bag. The only thing that can make changes the

location of its strands is if `getter`(G) executes, say by function H. At this point, H (which is currently active) becomes the operating function for both G and F — therefore, F is now confluent. All strands of F (and incidentally G) move to H's S bag.

Case 2: F is confluent and G is its (active) operating function; by inductive hypothesis, all strands of F are in G's S bag. The only thing that changes the location of F's strands is if G returns. At this point G becomes non-confluent (since it is no longer active); therefore F also becomes non confluent. All of F's strands move to G's P bag. □

We are now ready to prove theorem 7.1. The intuition is that if a function F is confluent, then there is some strand w in its (active) operating procedure which is a join ancestor of all strands of F and a spawn ancestor of currently executing strand.

Proof of Theorem 7.1. By lemma 7.5, we know that if $u \prec v$, then we can find a node w such that u is a join predecessor of w and w is a spawn predecessor of v. By property 7.2, since v is executing, the function containing w, say G, is still active. Therefore, by definition, the function containing u is confluent. Therefore, by lemma 7.6, u is an S bag.

If u does not precede v, then there is no path from u to v. Therefore u can not have a path to any strand w in any active function (otherwise by the second statement of property 7.2, since w has path to v, u will also have a path to v). Therefore, by definition, u is non confluent. By lemma 7.6, u is in a P bag. □

7.3 Full Race Detection Algorithm

In the previous section, we discussed how to maintain reachability data structure dynamically. We now discuss how this algorithm can be used to do race detection in a program with

structured futures. In order to do so, we must consider the other aspect of race-detection, namely the **access history** — for each memory location ℓ, the access history maintains enough information about the previous accesses to ℓ so that future accesses to ℓ can detect races.

For serial race detection of series-parallel programs, the access history for each memory location ℓ contains the last *serial* reader strand r and writer strand w for ℓ [71]. Whenever a strand s reads from a memory location ℓ, the race detector checks the reachability data structure to determine whether s is logically parallel with the last writer w; if so, a race is reported. Otherwise, the detector checks if s is in series with the last reader, and replaces it if so. Crucially, storing only the last serial reader suffices when the computation is series-parallel. A writer s is similar but compared against both the last reader and writer. For parallel race detection of series-parallel programs, it suffices to maintain two readers and one writer [132]. In both cases, the access history stores a constant number of previous accesses, and each memory access leads to at most a constant number of queries into the reachability data structure.

This property no longer holds for programs with futures, however. In particular, the access history for memory location ℓ still holds only one writer strand, namely the most recent writer `last-writer`(ℓ). However, it must now store an arbitrarily large `reader-list`. Race detection proceeds as follows. Whenever a strand s reads from the memory location ℓ, the race detector checks the reachability data structure to determine whether s is logically parallel with `last-writer`(ℓ); if so, a race is reported. Otherwise, s is added to `reader-list`(ℓ). When a strand s writes to a memory location ℓ, the race detector must check s against all readers in `reader-list`(ℓ) and with `last-writer`(ℓ). If s is in parallel with any of them, then it declares a race. Otherwise, the reader list is set to `empty` and s is stored as `last-writer`(ℓ).

1. Currently executing strand s reads location ℓ: check if `last-writer`(ℓ) is in a P bag; if so, declare a race. Otherwise, append r to `reader-list`(ℓ)
2. Currently executing strand s writes to location ℓ: check if any reader $r \in$ `reader-list`(ℓ) is in a P bag; if so, declare a race. Otherwise, empty the reader list and set `last-writer`$(\ell) = w$.

Figure 7.1: Pseudocode for memory accesses, using the extended SP-Bags algorithm to perform reachability queries.

We can empty the reader list without missing any races, because anything that executes later that would be in parallel with these readers must also be in parallel with s (which is the new `last-writer`(ℓ) and the race will be reported with s. Figure 7.1 shows the pseudocode for each memory access.

Unlike series-parallel computations, each write may generate multiple queries. However, we can bound the total number of queries since each writer removes the entire `reader-list`, yielding the following theorem.

Theorem 7.7. *The total running time of race detection for structured single-touch future-parallel programs is $O(T_1\alpha(m, n))$ time, where α is the inverse Ackermann's function, m is the number of memory accesses, and n is the number of* `spawn` *or* `create_future` *calls in the program.*

Proof. The fast disjoint-sets data structure provides the bound of amortized time $O(\alpha(m, n))$ per operation, where m is the number of operations and n is the number of sets. For our program, m is at most the number of memory accesses and n is the number of strands in the program. Clearly we only do T_1 Make-Set and Union operations, so we only need to bound the number of Find operations, which occur at queries.

We only do queries at memory accesses. On each read, we do one query — checking the reachability between the last writer and the current strand. On each write, we may do many queries — against all of the readers in the reader list. However, we remove all the readers in the `reader-list` are removed at a write. Therefore, each read leads to at most two queries, one when the read itself occurs and another when a subsequent write to the same memory location occurs. So the total number of queries is bounded by $2 \times$ number of reads. Since the total number of reads is at most T_1, the total number of disjoint-sets operations is $O(T_1)$.

The number of sets is the total number of places at which parallelism is created, i.e. the number of `spawn` or `create_future` calls in the program. Thus the total cost of race detection is $O(T_1\alpha(T_1, n))$. □

7.4 Related Work

As discussed in section 4.5, there is a large body of work on race detection for fork/join programs. Other structured computations have also been considered; Dimitrov et al.[60] propose an algorithm for race detection on computations that look like grids while Lee and Schardl [119] propose a race detector for fork-join computations that use a special kind of reduction mechanism. Recently, Surendran and Sarkar [173] proposed the first race detection algorithm for programs that use futures. Their reachability data structure has significantly more overhead than ours, however; in particular, the running time increases quadratically with the number of futures (that is multiplicatively instead of additively as in our case). There are two important distinctions between our approaches. First, the reachability data structure does not encode paths that include both SP and non-SP edges. Therefore, to answer a single reachability question of whether $u \prec v$, they must make multiple queries to the reachability data structure. Second, their reachability data structure explicitly stores

a dag and each reachability query does a search on the dag; therefore, each query to the reachability data structure can take more than constant time.

In addition to race-detection for programs with structured parallelism and futures, there is a rich literature on dynamic race detection for programming models that generate computations with nondeterministic dependence structures, such as ones that involve locks [164, 45, 153, 47, 141, 191, 152, 72, 69, 55]. For such models, since the output necessarily depends on the schedule, the best correctness guarantee that a race detector can provide is for a given program, for a given input, and for a given *schedule*. Also, these race detectors are often based on a persistent threading model; applying them to dynamically multithreaded programs means tracking all memory accesses of the work-stealing scheduler, resulting in high overhead and reporting of benign races in the runtime system.

7.5 Conclusions and Future Work

This chapter presented a race-detection algorithm for programs that use futures in a structured way. This structured use of futures still admits many useful applications [92], but our race detection algorithm to be very efficient. The overhead for our algorithm is proportion to the inverse Ackermann's function, which is so slow-growing that for all intents and purposes the bound is asymptotically optimal.

One obvious avenue for future work is to implement this algorithm and evaluate them, using the runtime from chapter 5 to implement futures. Also, our algorithm only works when the computation is executed serially — it would be interesting to consider how to parallelize the algorithm. There may be larger classes of restricted futures that still admit efficient race detection algorithms. Chapter 8 discusses race detection for arbitrary use of futures.

Chapter 8

Efficient Race Detection for General Future-Parallel Computations

The previous chapter presented an efficient (nearly asymptotically-optimal) algorithm for detecting races in computations that use futures in a restricted way. In this chapter we consider race detection on general future-parallel computations, placing no restrictions on how futures can be used. This generality comes at a cost, though we argue that this cost is reasonable for many programs.

General use of futures can generate computation DAGs with arbitrary dependencies. Therefore it seems unlikely that we can provide race detection without some asymptotic overhead. We present an algorithm whose running time depends on the number of `get_future` operations performed in the computation. In particular, we reduce the race-detection problem to doing on-the-fly dynamic reachability queries on a series-parallel dag with extra edges due to `get_future`. Our algorithm runs in $O(T_1 + k^2)$ time, where k is the number of `get_future` calls in the computation.

As noted in chapter 7, `create_future` and `get_future` are strictly more general than `spawn` and `sync`, since we can replace each `spawn` with a `create_future` and each `sync` with a `get_future` on each of the futures created so far. However, we do not do so in the model for this chapter, since replacing `sync` operations with `get_future` will increase k. Separating the two types of parallel primitives is in fact a major advantage of our model.

This dynamic reachability algorithm can be extended with an access history component using the same approach as chapter 7, resulting in a full determinacy race detection algorithm that runs in time $O(T_1 + k^2)$.

Outline This is chapter is structured as follows. Section 8.1 explains how we can utilize our knowledge of series-parallel DAGs to model general DAGs, while section 8.2 builds on this to explain an offline reachability algorithm. We extend this to an online algorithm in section 8.3. The algorithm is proven correct in section 8.4 and the performance of the full race detection algorithm is analyzed in section 8.5. Sections 8.6 and 8.7 discuss related work and conclude, respectively.

8.1 "Nearly" Series-Parallel DAGs

We model future-parallel computations as a series-parallel DAG with some **non-SP** edges. The `create_future` primitive creates a sub-SP-dag which is ended by the `get_future` primitive. We add artificial SP edges from the computation's root to each future task's start node, and from each future task's end node to the computation sink. These edges are for modeling purposes only, allowing us to support future tasks that create and get other futures. These additional edges do not change the meaning of the DAG and are less restrictive than create/get edges.

Note that the root of the computation may now have outdegree greater than two. As in previous chapters, we would like to assume binary forking and spawning. So we remedy this by adding a binary tree of height $\log k$ to the start of the computation, where k is the number of future tasks in the computation.

This model of series-parallel computation dags with arbitrary non-SP edges is quite general and subsumes computations that can arise from future [127, 40, 73, 43, 42, 110, 9, 86] or other future-like (such as "put" and "get" [37, 180]) parallel constructs that we encountered in the literature. Therefore, our algorithm would work on all of these primitives.

We need to answer reachability queries in such graphs, where $O(k)$ arbitrary non-SP edges have been added. We will show how to build a data structure in time $O(n + k^2)$, where n is the total number of vertices.

Formally, we consider reachability on a graph $G = (V, E_{SP} \cup E_{\text{non}})$. The minor $G_{SP} = (V, E_{SP})$ contains all the series-parallel edges, while E_{non} contains the non-SP edges. We assume non-SP edges are not incident on fork or join nodes and that the computation's source node is not a fork node[1]. We also assume that the graph description specifies which edges are part of E_{SP} and which are the extra edges of E_{non}. This information is provided in a programming model with different linguistic keywords for different edge types — both `spawn` and `sync` and `create_future` and `get_future`. This has a minor disadvantage to the programmer, since now they must remember more keywords. However, we will see that it allows our reachability data structure to be much more efficient.

We write $u \prec v$ to denote the presence of a directed path from u to v in the graph in question. We say that u is a **predecessor** of v and v is a **successor** of u. To disambiguate, we often

[1]This is without loss of generality, since we can add a constant number of nodes to fix any violations.

indicate the graph being discussed as a subscript of the precedes symbol, e.g., $u \prec_G v$ to mean that there is a path from u to v in G. (The path can be empty, i.e., we always have $v \prec v$.) We use $u \prec_{\text{SP}} v$ as a shorthand for $u \prec_{G_{\text{SP}}} v$, i.e., to indicate that there is a path from u to v using only the edges E_{SP}. We use $u \rightsquigarrow v$ to refer to a (possibly empty) directed path from u to v, and we use $u \overset{G}{\rightsquigarrow} v$ to mean a path in G. As before, we use $u \overset{\text{SP}}{\rightsquigarrow} v$ as a shorthand for $u \overset{G_{\text{SP}}}{\rightsquigarrow} v$.

Consider a node v. If x and y are both predecessors of v and $x \prec y$, then we say that y is a **nearer** predecessor of v. Similarly, if x and y are both successors of v, and $x \prec y$, then we say that x is a **nearer** successor. When we use the term "nearer" in this section, we always mean with respect to the series-parallel graph G_{SP}.

8.2 Offline Reachability

For ease of exposition, we first present how to compute reachability given a DAG *a posteriori*. We will prove this technique correct computes reachability, then show an online technique that maintains these same data structures for the currently exposed nodes. We should note that we do not know of any offline algorithm with better bounds than this algorithm.

It is instructive to first consider a very simple approach for computing reachability statically: perform a graph search from each node, building the transitive closure. Given a computation DAG $G = (V, E)$, This can extended to an online algorithm by storing the set of predecessors at each vertex. When we add a new edge (u, v), we check if v has gained any new predecessors by comparing the predecessor sets of u and v. If v gains a new predecessor, we update the predecessors for v's neighbors. Note that a vertex can gain a new predecessor at most $|V|$

times, yielding a total time of $O(|V||E|)$ and supporting queries in $O(1)$ time. In the worst case we have $|E| = \Theta(T_1)$, so this method runs in time $O(T_1^2)$.

Rather than explicitly computing reachability on the entire computation DAG, we will utilize the model from section 8.1 to allow us to keep two smaller graphs, along with a reachability data structure for each. One of these graphs, $G_{\text{SP}} = (V, E_{\text{SP}})$, will contain all the nodes of the computation graph, but only series-parallel edges. The other, $\mathcal{R} = (V_{\mathcal{R}}, E_{\mathcal{R}})$, is an auxiliary graph which is not series-parallel.

We will leverage prior work on series-parallel reachability for the graph G_{SP} and simply build the transitive closure for $\mathcal{R}$. When we need to query reachability between two nodes u and v, we first query the SP reachability data structure. If this reports a serial relationship, we are done. Only if it reports a parallel relationship do we query the second, non-SP reachability data structure. This is the key idea that allows our algorithm to be efficient: since we $\mathcal{R}$ for *all* reachability queries, we add only a select subset of nodes to it. So building this data structure is fast, even though we compute reachability on $\mathcal{R}$ in a very simple way.

Bender et al. [19] addressed how to efficiently build a reachability data structure for G_{SP} (summarized and extended to parallel execution in chapter 4). In particular, the data structure can be constructed in $O(|V|)$ time and support queries in $O(1)$ time. (SP-Order is overkill for the offline setting, but also applies in the dynamic case.) Thus the focus of this work is on the efficient construction of $\mathcal{R}$.

8.2.1 Properties of the Auxiliary Graph

Here we describe the key features of the auxiliary graph $\mathcal{R}$. We will later consider the specifics in more detail, but these properties are enough to imply correctness of the query. Moreover, these properties are the core motivations of algorithm design (both static and dynamic).

The vertices in $\mathcal{R}$ are a subset of nodes from the original graph. We call these nodes $V_{\mathcal{R}} \subseteq V$ the **anchor nodes**. The set of anchor nodes comprises all nodes incident on the non-SP edges E_{non}, as well as some (but not too many) other nodes. The graph $\mathcal{R}$ is designed to ensure the following three properties, the third motivating the extra anchor nodes:

Property 8.1. For any two anchor nodes $u, v \in V_{\mathcal{R}}$, we have $u \prec_{\mathcal{R}} v$ if and only if $u \prec_G v$.

Property 8.2. $|V_{\mathcal{R}}| = O(|E_{\text{non}}|)$ and $|E_{\mathcal{R}}| = \Theta(|E_{\text{non}}|)$.

We associate with each node v in G two specific anchor nodes (possibly null): an **anchor predecessor** and **anchor successor**, denoted $\mathit{PredAnchor}(v)$ and $\mathit{SuccAnchor}(v)$, respectively. The key property of these anchor predecessor and successors is as follows.

Property 8.3. First, if not null, the anchor predecessor (or successor) is a predecessor (or successor) in G_{SP}, i.e., for every node v there exists a path $\mathit{PredAnchor}(v) \overset{\text{SP}}{\rightsquigarrow} v \overset{\text{SP}}{\rightsquigarrow} \mathit{SuccAnchor}(v)$.

Consider any non-SP edge $(x, y) \in E_{\text{non}}$. If there is a path $x \to y \overset{\text{SP}}{\rightsquigarrow} v$, then $\mathit{PredAnchor}(v)$ is not null and there exists a path of the form $x \to y \overset{\text{SP}}{\rightsquigarrow} \mathit{PredAnchor}(v) \overset{\text{SP}}{\rightsquigarrow} v$. Similarly, a path $v \overset{\text{SP}}{\rightsquigarrow} x \to y$ implies a path $v \overset{\text{SP}}{\rightsquigarrow} \mathit{SuccAnchor}(v) \overset{\text{SP}}{\rightsquigarrow} x \to y$.

In other words, $\mathit{PredAnchor}(v)$ is nearer than all other predecessors that have incoming non-SP edges. (It may be null if there are no predecessors with incoming non-SP edges.)

Similarly, *SuccAnchor*(v) is nearer than all other successors having outgoing non-SP edges. (It is null if there are no successors with outgoing non-SP edges).

8.2.2 Reachability Queries

Assuming an auxiliary graph with the aforementioned properties, querying whether $u \prec_G v$ operates as follows:

PRECEDES(u,v)

1 **if** $u \prec_{\text{SP}} v$ // Query the reachability structure on G_{SP}

2 **return** TRUE

3 **elseif** *SuccAnchor*(u) $\neq$ NULL, *PredAnchor*(v) $\neq$ NULL, and
SuccAnchor(u) $\prec_{\mathcal{R}}$ *PredAnchor*(v)
// Query the reachability structure on $\mathcal{R}$

4 **return** TRUE

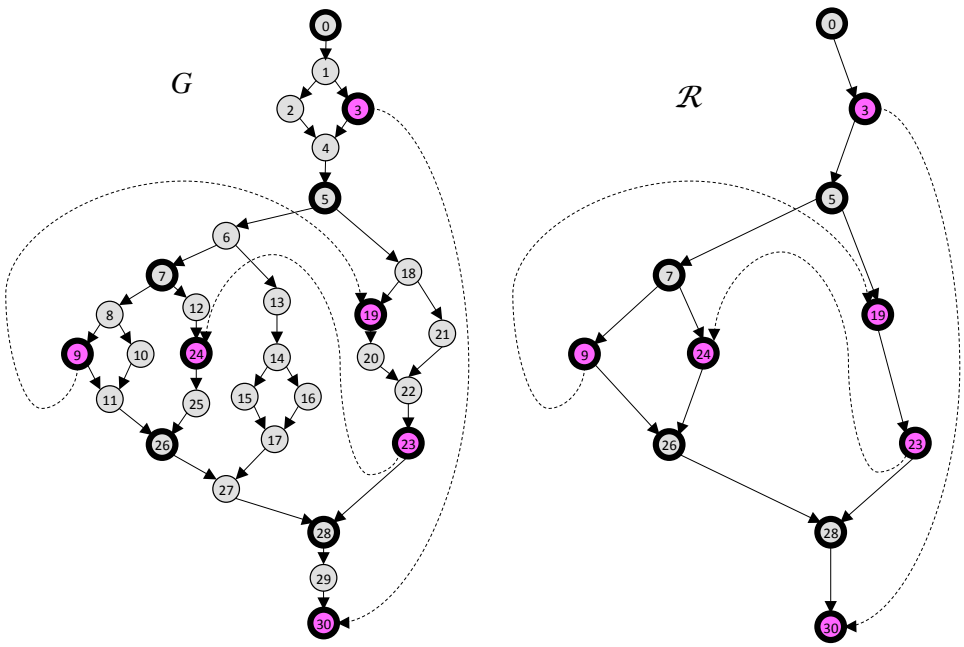
5 **else return** FALSE

Note that we only query the reachability structure on $\mathcal{R}$ if $u \not\prec_{\text{SP}} v$ — this is important for correctness; the second query may return the wrong answer if $u \prec_{\text{SP}} v$.

The following lemma says that the query is correct.

Lemma 8.4. *Assuming properties 8.1 and 8.3, the query algorithm correctly returns $u \prec v$ if and only if there is a path from u to v in G.*

Proof. There are three cases, corresponding to each of the returns. The first case is that $u \prec_{\text{SP}} v$. Then the algorithm trivially returns the correct answer (true).



node	anchor pred.	anchor succ.
0	0	0
1	0	3
2	0	5
3	3	3
4	3	5
5	5	5
6	5	7
7	7	7
8	7	9
9	9	9
10	7	26
11	9	26
12	7	24
13	5	28
14	5	28
15	5	28
16	5	28
17	5	28
18	5	19
19	19	19
20	19	23
21	5	23
22	19	23
23	23	23
24	24	24
25	24	26
26	26	26
27	26	28
28	28	28
29	28	30
30	30	30

Figure 8.1: Example graph G (left), auxiliary graph $\mathcal{R}$ (middle), and the corresponding anchor predecessors/successors (right). The dashed arrows correspond to the non-SP edges E_{non}; omitting the dashed (non-SP) edges from the graphs yields the series-parallel subgraphs G_{SP} and $\mathcal{R}_{\text{SP}}$, respectively. Nodes with thicker borders are the anchor nodes, and the magenta nodes are the principle anchors (those incident on non-SP edges). The nodes are numbered by their execution order.

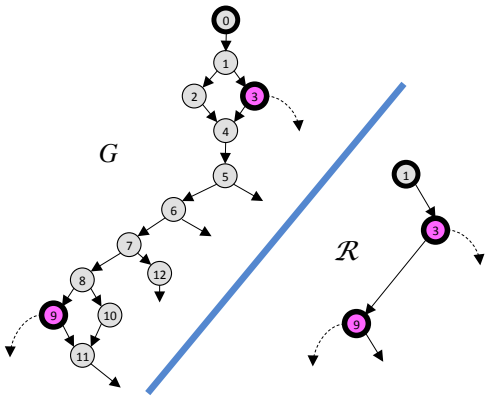


Figure 8.2: A partial execution of the dag from figure 8.1 just after node 12 has been executed. Only nodes that have been processed are displayed.

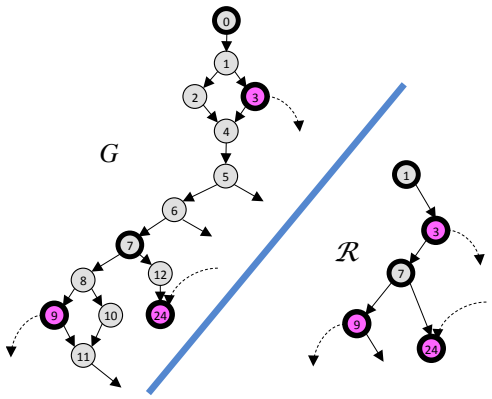


Figure 8.3: A partial execution of the dag from figure 8.1 just after processing node 24. Note that 24 cannot execute because it has an unsatisfied incoming non-SP edge, so the node now blocks and the execution would continue with 6's other child.

The second case is that $u \not\prec_{\text{SP}} v$ and $u \prec_G v$. Let $p = u \overset{G}{\rightsquigarrow} v$ be any path from u to v. Since there is no path in G_{SP}, p must use at least one non-SP edge. If there is just one non-SP edge (a, y) on the path, then we can rewrite the path as $u \overset{\text{SP}}{\rightsquigarrow} a \xrightarrow{E_{\text{non}}} y \overset{\text{SP}}{\rightsquigarrow} v$. Otherwise, let (a, b) be the first non-SP edge on the path, and let (x, y) be the last non-SP edge on the path. Then $p = u \overset{\text{SP}}{\rightsquigarrow} a \xrightarrow{E_{\text{non}}} b \overset{G}{\rightsquigarrow} x \xrightarrow{E_{\text{non}}} y \overset{\text{SP}}{\rightsquigarrow} v$. In either case, since a has an outgoing non-SP edge, we can apply property 8.3 to conclude that the path $u \overset{\text{SP}}{\rightsquigarrow} \mathit{SuccAnchor}(u) \overset{\text{SP}}{\rightsquigarrow} a$. Similarly, there also exists a path $y \overset{\text{SP}}{\rightsquigarrow} \mathit{PredAnchor}(v) \overset{\text{SP}}{\rightsquigarrow} v$. Splicing these paths together appropriately with p, we conclude that there exists a path $p' = u \overset{\text{SP}}{\rightsquigarrow} \mathit{SuccAnchor}(u) \overset{G}{\rightsquigarrow} \mathit{PredAnchor}(v) \overset{\text{SP}}{\rightsquigarrow} v$. Importantly, there is a path $\mathit{SuccAnchor}(u) \overset{G}{\rightsquigarrow} \mathit{PredAnchor}(v)$. Thus, by property 8.1 $\mathit{SuccAnchor}(u) \prec_{\mathcal{R}} \mathit{PredAnchor}(v)$ and the query correctly returns true.

Finally, suppose $u \not\prec_G v$. We claim that $\mathit{SuccAnchor}(u) \not\prec_G \mathit{PredAnchor}(v)$, and hence by property 8.1 $\mathit{SuccAnchor}(u) \not\prec_{\mathcal{R}} \mathit{PredAnchor}(v)$ and the query correctly returns false. We justify the claim by contradiction—if there exists a path $\mathit{SuccAnchor}(u) \rightsquigarrow \mathit{PredAnchor}(v)$, then by property 8.3 there also exists a path $u \rightsquigarrow \mathit{SuccAnchor}(u) \rightsquigarrow \mathit{PredAnchor}(v) \rightsquigarrow v$, and hence $u \prec_G v$, which is a contradiction. □

8.2.3 The Auxiliary Graph

Now we discuss the construction of the *a posteriori* auxiliary graph. In reality, this graph is built-up incrementally while executing G. But since nodes are never removed, it is helpful to reason about the auxiliary graph as a static entity. An example graph and auxiliary graph are shown in figure 8.1.

As already noted, $V_{\mathcal{R}}$ includes all of the vertices in G that are incident on non-SP edges. We also add to $V_{\mathcal{R}}$ just enough fork and join nodes to make property 8.3 feasible. In particular, the anchor nodes include all of the following:

- all nodes incident on a non-SP edge, which we call **principle anchors**,
- the first node $source(G_{\text{SP}})$ of the entire graph (to remove some corner cases),
- all fork nodes whose left and right subdags in G_{SP} each contain at least one principle anchor, and
- all join nodes whose left and right subdags in G_{SP} each contain at least one principle anchor.

In the example, fork node 7 and corresponding join node 26 are both anchor nodes because each of the left and right subdags contain a principle anchor, namely 9 and 24, respectively. In contrast, fork 6 and corresponding join 27 are not anchor nodes because only the left subdag contains any principle anchors.

Note that which nodes are anchors depends only on the series parallel graph G_{SP} and which nodes have incident non-SP edges. How the nodes are related by non-SP edges, as well as the direction of the edges,does not affect anything except specific edges in $\mathcal{R}$.

The reason for making certain join nodes anchors is to make it possible to define anchor predecessors consistent with property 8.3. Otherwise, node 29 could not possibly have an anchor predecessor, as there is no anchor node that comes after both 19 and 24. Similarly, certain forks are made into anchors to enable an anchor successor.

But we also have to be careful not to make $\mathcal{R}$ too large. By making forks and joins anchors only when both subdags have principle anchors, we are able to bound the number of anchor nodes. Proof of the following lemma follows the same ideas as bounding the number of internal nodes in a full binary tree with respect to the number of leaves. The "tree" here is nesting subdags of anchored forks.

Lemma 8.5. *There are $O(|E_{\text{non}}| + 1)$ anchor nodes (which are exactly the vertices in $\mathcal{R}$).*

Proof. Let p be the number of principle anchors. There are (at most) two principle anchors induced by each non-SP edge, so we have $p \leq 2\,|E_{\text{non}}|$. If $p = 0$, the only anchor is the first node in the graph. The rest of this lemma considers the case that $p \geq 1$.

To count the total number of anchors, we consider a collection of series-parallel graphs inductively over the series/parallel compositions used to create G_{SP}, starting from the base case singleton nodes. The idea is to count the number x of graphs containing principle anchors. Initially, $x = p$ as each principle anchor is in a separate graph. This number can only decrease when graphs are composed, since compositions do not create *principle* anchors.

The key observation is that whenever a composition rule results in more (fork or join) anchors, the number x of graphs with principle anchors decreases by one. In particular, anchors are only created on parallel composition of two graphs having principle anchors. In this case, 2 new anchor nodes are created. But two graphs with principle anchors are combined into one, so x decreases by 1. Thus, this case can occur at most $p - 1$ times overall. No other cases results in anchor nodes being created, so the total number of anchor nodes including the source node is at most $p + 2(p - 1) + 1 = 3p - 1$. □

Defining the anchor predecessor and successor. There is some flexibility in how we choose the anchor predecessor and successor of each node u in G_{SP}, as there may be multiple nodes that meet the requirements of property 8.3. We choose the nearest such node:

Definition 8.6. The anchor predecessor of v, denoted $\mathit{PredAnchor}(v)$, is the node y such that (1) $y \in V_{\mathcal{R}}$ is an anchor node, (2) $y \prec_{\text{SP}} v$ is a predecessor of v in G_{SP}, and (3) y is nearer, with respect to G_{SP}, than all other anchor nodes preceding v; that is, for all other

anchor nodes $x \prec_{\text{SP}} v$, we have $x \prec_{\text{SP}} y$. Since the first node of the graph is an anchor node, the anchor predecessor is never null.

The anchor successor is defined symmetrically except that it may be null if v has no successors that are anchor nodes.

It should be straightforward to see that, if the above definition is well-defined, i.e., if such a node always exists, then this definition of anchor predecessor and anchor successor satisfies property 8.3. What may or may not be obvious is that the definition is well-defined.

Lemma 8.7. *The definition of anchor predecessor (or successor) is well-defined. That is to say, for every node with any predecessors (or successors) that are anchors, there is exactly one such anchor node that is nearer than all others. Thus, the conditions of property 8.3 are satisfied.*

Proof. Consider the anchor predecessor. (The argument for successor is similar.) Suppose for the sake of contradiction that there exists some node v for which the *PredAnchor*(v) is ill-defined, i.e., there is no node meeting the requirements. Then there must be at least two distinct anchor nodes $x_1, x_2 \prec_{\text{SP}} v$ such that: there is no anchor node closer to v than x_1 or x_2. Consider any paths $p_1 = x_1 \overset{\text{SP}}{\rightsquigarrow} v$ and $p_2 = x_2 \overset{\text{SP}}{\rightsquigarrow} v$. Let u be the earliest node at which these paths cross (and possibly $u = v$). Then u must be a join node having anchor nodes in both of its subdags, so u would be an anchor node and $x_1, x_2 \prec_{\text{SP}} u$. This contradicts the assumption that there is no anchor node nearer to v than x_1 or x_2. □

The edges in $\mathcal{R}$. The edges in $E_{\mathcal{R}}$ consist of all of the non-SP edges E_{non} plus just enough edges to ensure that the anchor nodes have the same transitive closure in both G and $\mathcal{R}$ (property 8.1). Specifically, $\mathcal{R}$ consists of a series-parallel minor $\mathcal{R}_{\text{SP}} = (V_{\mathcal{R}}, E_{\mathcal{R}})$ of G_{SP},

plus the non-SP edges E_{non}. Moreover, an anchor node is a fork or join in $\mathcal{R}_{\text{SP}}$ if and only if it is also a fork or join in G_{SP}.

For each anchor, its edges in $\mathcal{R}_{\text{SP}}$ are:

- For a non-join anchor node v, let (u, v) be the only incoming edge in G_{SP}. Then v has exactly one incoming edge in $\mathcal{R}_{\text{SP}}$: the edge $(\mathit{PredAnchor}(u), v)$.
- For a join anchor node j, let (ℓ, j) and (r, j) be the two incoming edges in G_{SP}. Then j has exactly two incoming edges in $\mathcal{R}_{\text{SP}}$: the edges $(\mathit{PredAnchor}(\ell), j)$ and $(\mathit{PredAnchor}(r), j)$.
- For a fork anchor node f, let (f, ℓ) and (f, r) be the two outgoing edges in G_{SP}. Then f has exactly two outgoing edges in $\mathcal{R}_{\text{SP}}$: the edges $(f, \mathit{SuccAnchor}(\ell))$ and $(f, \mathit{SuccAnchor}(r))$.

With the exception of the black-box reachability structure and making nodes into principle anchors, our algorithm entirely ignores E_{non}, so we will generally reason about $\mathcal{R}_{\text{SP}}$. We note that the only two black-box reachability structures are with respect to G_{SP} and $\mathcal{R}$; we never perform reachability queries on $\mathcal{R}_{\text{SP}}$ (after all, these would return the same answer as queries in G_{SP} but restricted to anchor nodes).

Lemma 8.8. *This definition of $\mathcal{R}$ is well-defined. Moreover, it satisfies property 8.1. That is, for any two anchor nodes $u, v \in V_{\mathcal{R}}$, we have $u \prec_{\mathcal{R}} v$ if and only if $u \prec_G v$.*

Proof. We start by showing that $\mathcal{R}$ is well defined. The only issue is with respect to the forks. We need to verify (1) that we do not add any edges of the form (f, NULL), and (2) that there are exactly two outgoing edges. The latter is not immediately obvious given that

edges are also defined in the other direction. (1) follows from the fact that a fork is only an anchor if both subdags have anchors, and hence by lemma 8.7 the sources ℓ and r of each subdag has an anchor successor. Moreover, for (2), $\mathit{SuccAnchor}(\ell)$ and $\mathit{SuccAnchor}(r)$ are "between" ℓ and r, respectively, and all other succeeding anchors. Thus, no other anchor would an edge originating at f.

We next show property 8.1.

($\Rightarrow$) We will show the contrapositive. By definition, the anchor predecessor of a node precedes that node in G_{SP}. Similarly, the anchor successor is a successor of the node. By inspection of edges added, $\mathcal{R}_{\text{SP}}$ only includes edges between nodes that have the proper relationship in G_{SP}. That is to say, if $u \not\prec_{G_{\text{SP}}} v$ then $u \not\prec_{\mathcal{R}_{\text{SP}}} v$.

($\Leftarrow$) Suppose $u \prec_G v$. Then there exists a path p from u to v. Choose p to be the path that goes through the most possible anchors, and partition it at every anchor; that is, $p : u = x_0 \overset{G}{\rightsquigarrow} x_1 \overset{G}{\rightsquigarrow} \cdots \overset{G}{\rightsquigarrow} x_{k-1} \overset{G}{\rightsquigarrow} x_k = v$. We claim that for each subpath, $p_i : x_i \prec_{\mathcal{R}} x_{i+1}$, which is enough to imply $u \prec_{\mathcal{R}} v$. To prove the claim, suppose first that the $x_i \rightsquigarrow x_{i+1}$ contains a non-SP edge. Then it consists of a single edge $x_i \xrightarrow{E_{\text{non}}} x_{i+1}$ because both endpoints of non-SP edges are anchors. Otherwise, $x_i \overset{\text{SP}}{\rightsquigarrow} x_{i+1}$. By choice of p, x_i must be the anchor predecessor of the node that immediately precedes x_{i+1} on the path. (Or else there is a longer path.) Thus, the edge $(x_i, x_{i+1}) \in \mathcal{R}_{\text{SP}}$. □

8.3 Online Construction of Reachability Data Structures

This section outlines issues relating to constructing the data structure efficiently while executing the graph sequentially. Our algorithm proceeds through the following steps: select a node u that has not yet executed; **process u**, by which we mean update the data structures;

if all of u's predecessors have been executed, **execute** $\boldsymbol{u}$, which corresponds to executing the original instructions in the program.[2] After executing u, the outgoing edges from u and corresponding nodes are revealed to the algorithm and may be selected. Initially, only the source node is available to select.

The main algorithm is in processing the nodes, which is where all the data-structural updates occur. In particular, we need to: keep track of the anchor predecessor; maintain the anchor successor; add nodes and edges G_{SP}, $\mathcal{R}$, and $\mathcal{R}_{\text{SP}}$ as appropriate; and update both reachability structures. The reachability structures are a black box implied by the underlying graphs G_{SP} and $\mathcal{R}$, so it suffices to specify when vertices and edges are added. The bulk of this section is devoted to discussing this processing step.

Note that the anchors, anchor predecessor, anchor successor, and $\mathcal{R}$, are all defined as in section 8.1 but with respect to the subgraph of nodes processed thus far. We maintain $\mathcal{R}_{\text{SP}}$ and $\mathcal{R}$ explicitly, but we shall describe only $\mathcal{R}_{\text{SP}}$—$\mathcal{R}$ is formed by adding the non-SP edges E_{non}.

As it turns out, maintaining the anchor predecessor is relatively easy, but keeping track of the anchor successor is more complicated. Our algorithm does not keep the anchor successor directly, instead using a level of indirection. For performance, our algorithm requires a specific depth-first execution order, discussed next.

8.3.1 Execution Order

Where possible, our algorithm selects nodes in the dag in depth-first, left-to-right order with respect to the series-parallel graph G_{SP}. That is to say, we always process and execute as

[2]Executing a node is where the race detector would perform queries into the data structure.

much of the left subdag of a fork node as possible before starting to process the right subdag of that fork. If there are no non-SP edges, this would mean executing the left subdag entirely before starting the right subdag, completing both before continuing to the join. However, because nodes are only ready to execute after their predecessors, we may have to delay certain nodes due to dependencies on non-SP edges. We call such delays **blocking**. A blocked node is executed whenever its dependencies are satisfied.

For concreteness, the execution operates as follows. Let S denote the depth-first, left-to-right sequential ordering of G_{SP} (i.e., ignoring any dependencies in E_{non}). On each step of the algorithm select the unexecuted, unblocked node u that is earliest in S.

For example, the nodes in figure 8.1 are labeled according to their execution order in our algorithm. After executing node 12, we process the node 24. But node 24 cannot finish executing due to a dependency on 23. As such, 24 blocks and we continue with node 13 in depth-first order. After 23 is executed, 24 unblocks and can we resume the depth-first execution from there.

8.3.2 Challenges

The biggest challenge is that when a fork node is first encountered, we do not know if it will be an anchor. A fork node only becomes an anchor when a principle anchor has been processed in both of the fork's subdags. Dealing with joins is easier because both subdags have executed completely before the corresponding join is processed, so we know immediately whether a join should be an anchor node.

Forks becoming anchors has a few ramifications. First, we need to verify that a fork node f can be spliced into the right place in $\mathcal{R}_{\text{SP}}$. Second, arbitrarily many nodes may now have f

as their anchor predecessor. Third, arbitrarily many nodes may now have f as their anchor successor. The worry is that all of these updates would be too expensive.

Consider, for example, the partial execution shown in figure 8.1, just following node 12's execution. Next, node 24 is processed (but not executed because it blocks), so the structures should be updated as in figure 8.3. Many changes have occurred due to the discovery of this new principle anchor. The fork node 7 becomes an anchor node, so it must be added to $\mathcal{R}_{\text{SP}}$, spliced in between 3 and 9. The newly discovered principle anchor 24 must also be added to $\mathcal{R}$, but this case is easier because it hangs off the end of the graph. In addition, many nodes have a new anchor predecessor or anchor successor. Specifically, the anchor predecessor of 7, 8, 10 and 12 changes from 3 to 7. The anchor successor of 2, 4, 5, 6, and 7 changes from 9 to 7. And the anchor successor of 12 changes from NULL to 24.

8.3.3 Anchor Predecessor/Successor and Proxies

We store the anchor predecessor explicitly and update it through a graph search when creating any new anchors. We shall argue that the anchor predecessor cannot change more than a constant number of times due to the depth-first execution order.

Maintaining anchor successor explicitly, however, would be too expensive. Consider again our running example. The anchor successor of 2, 4, and 5 changes when 7 becomes an anchor. If we look at the *a posteriori* dag in figure 8.1, we can see that these will be updated again when 5 becomes an anchor. In fact, with a long chain of forks, the anchor successor may change a super-constant number of times.

Instead, we maintain the anchor successors through a level of indirection. For each node u, we store a **proxy** node $\mathit{proxy}(u)$, defined as follows:

1. If u's anchor successor is null, then $proxy(u) = \text{NULL}$.

2. If u is an anchor node, then $proxy(u) = u$.

3. If u has an anchor successor, and there exists a join node j satisfying all of the following three properties, then $proxy(u) = PredAnchor(j)$. The properties are (1) u is one of j's subdags, (2) that subdag does not contain any anchor nodes, and (3) the other subdag does contain at least one anchor node. (This is a case when j is not an anchor.)

4. Otherwise, $proxy(u) = PredAnchor(u)$.

In the example, $proxy(2) = proxy(4) = proxy(5) = proxy(6) = 3$ both before and after processing 24. Note that node 2 falls in the third case, whereas nodes 4, 5, and 6 fall in the fourth.

Given the proxy, we compute the anchor successor as follows. The idea is to look at the proxy's outgoing edge in $\mathcal{R}_{\text{SP}}$. If there is more than one, i.e., the proxy is a fork, then look in the direction that would lead to the node.

GetSuccAnchor(u)

1 **if** u is an anchor node

2 **return** u

3 **elseif** *proxy*(u) = NULL

4 **return** NULL

5 **else if** *proxy*(u) is not a fork node

6 **return** target of *proxy*(u)'s only out edge in $\mathcal{R}_{\text{SP}}$

7 **elseif** u is in *proxy*(u)'s left subdag in G_{SP}

// i.e., check if *proxy*(u).*left* $\prec_{G_{\text{SP}}} u$

8 **return** target of *proxy*(u)'s left out edge in $\mathcal{R}_{\text{SP}}$

9 **else return** target of *proxy*(u)'s right out edge in $\mathcal{R}_{\text{SP}}$

In our example, 5's proxy does not change after processing 24, but the target of the proxy's outgoing edge *does* change in $\mathcal{R}$. This is exactly why the indirection of the proxy helps us—updates to $\mathcal{R}$ implicitly capture the changes to the anchor successor. For example, *proxy*(5).*out* $= 9$ before processing 24, and *proxy*(5).*out* $= 7$ after processing 24. We can see that for this example at least, GetSuccAnchor(5) returns the correct answer. The following lemma says that it is correct in general.

Lemma 8.9. *Assuming* *proxy*(u) *is maintained as defined,* GetSuccAnchor(u) *correctly returns* *SuccAnchor*(u).

Proof. The cases that *proxy*(u) = NULL or u is an anchor are trivial.

(Case 3.) Suppose the *proxy*(u) is chosen according to the third (join-node) case. Then we claim that j and u have the same anchor successor. In particular, since u is in j's subdag, every path $u \overset{\text{SP}}{\rightsquigarrow} v$ to a successor v of j must pass through j. Thus, the only way u and j

can have different anchor successors is if there is an anchor node on a path $u \overset{\text{SP}}{\rightsquigarrow} j$. That contradicts the assumption that we fall in this case.

Moreover, j must fall into the fourth case, i.e., $\mathit{proxy}(j) = \mathit{PredAnchor}(j) = \mathit{proxy}(u)$. The reason is that (i) j has an anchor successor, (ii) one of j's subdags contains an anchor, eliminating case 3 for j, and (iii) j is not an anchor join because its other subdag does not have an anchor. Thus, finishing case 3 reduces to applying case 4 on j.

(Case 4.) Suppose $\mathit{proxy}(u) = \mathit{PredAnchor}(u)$. Let x be the returned value from GetSuccAnchor. If $x = \mathit{SuccAnchor}(u)$, we are done. Because $\mathcal{R}$ preserves reachability (lemma 8.8), we must have $\mathit{PredAnchor}(u) \prec_{\mathcal{R}} x \prec_{\mathcal{R}} \mathit{SuccAnchor}(u)$. By definition of anchor predecessor and anchor successor and correctness of $\mathcal{R}$, the only options possible are $x = \mathit{SuccAnchor}(u)$ and $u \not\prec_{\text{SP}} x, x \not\prec_{\text{SP}} u$. If $x = \mathit{SuccAnchor}(u)$, we are done, so suppose instead that x and u are not related. Then the paths from $\mathit{PredAnchor}(u)$ to u and x must diverge at some point, i.e., there must be some fork $f \neq \mathit{PredAnchor}(u)$ and corresponding join $j \neq \mathit{SuccAnchor}(u)$ with $\mathit{PredAnchor}(u) \prec_{\text{SP}} f \prec_{\text{SP}} x \prec_{\text{SP}} j \prec_{\text{SP}} \mathit{SuccAnchor}(u)$ and $\mathit{PredAnchor}(u) \prec_{\text{SP}} f \prec_{\text{SP}} u \prec_{\text{SP}} j \prec_{\text{SP}} \mathit{SuccAnchor}(u)$. Since the fork and join are not anchors, u's branch must not have an anchor node. This is a contradiction as it would make u fall in Case 3. □

8.3.4 Algorithm to Process Nodes

This section describes the algorithm for processing a node. Recall from section 8.2 that we explicitly construct $\mathcal{R}$ (adding nodes whenever new anchors are revealed), directly maintain $\mathit{PredAnchor}(u)$, and also maintain a value $\mathit{proxy}(u)$ that allows us to determine the anchor successor. This section describes the changes to these structures when processing a node. We do not describe the reachability structures here—when we insert an edge or vertex into $\mathcal{R}_{\text{SP}}$

or $\mathcal{R}$, we implicitly mean to update the reachability structure on $\mathcal{R}$ as a black box. Moreover, the query algorithm was already described in section 8.2.2, and the only remaining obstacle to correctness is ensuring that the algorithm correctly maintains what it is supposed to.

We break-up the description into multiple cases depending on the type of node being processed.

Case 1: Processing a node v that is neither a principle anchor nor a join. This is the easiest case. Let $(u, v) \in E_{\text{SP}}$ be the only incoming SP edge in G_{SP}. Set $\mathit{PredAnchor}(v) = \mathit{PredAnchor}(u)$ and $\mathit{proxy}(v) = \textsc{null}$. Note that this case captures v being a fork as well.

Case 2: Processing a join node j. Then there are two incoming edges $(\ell, j), (r, j) \in E_{\text{SP}}$, coming from the sinks of the j's left and right subdags in G_{SP}, respectively. Set $\mathit{proxy}(v) = \textsc{null}$. Next, we determine if j should become an anchor and set $\mathit{PredAnchor}(j) = p_j$, where p_j is computed as follows. Note that $u \prec_{\text{SP}} u$, so either of the first two cases covers equality:

$$p_j = \begin{cases} \mathit{PredAnchor}(r) & \text{if } \mathit{PredAnchor}(\ell) \prec_{\text{SP}} \mathit{PredAnchor}(r) \\ \mathit{PredAnchor}(\ell) & \text{if } \mathit{PredAnchor}(r) \prec_{\text{SP}} \mathit{PredAnchor}(\ell) \\ j & \text{otherwise} \end{cases} \tag{8.1}$$

If $p_j \neq j$, then we are done.

Otherwise ($p_j = j$), j is made into an anchor. Add the vertex j and the edges $(\mathit{PredAnchor}(\ell), j)$ and $(\mathit{PredAnchor}(r), j)$ to $\mathcal{R}_{\text{SP}}$. Update $\mathit{proxy}(j) = j$.

Finally, perform a backwards graph search in G_{SP} from ℓ and r, only visiting nodes x for which $\mathit{proxy}(x) = \text{NULL}$, i.e., those predecessors with no anchor successor. Set $\mathit{proxy}(x) = \mathit{PredAnchor}(\ell)$ for those nodes encountered searching back from ℓ, and $\mathit{proxy}(x) = \mathit{PredAnchor}(r)$ for nodes encountered when searching back from r.[3]

Case 3: Processing a (non-join) node v that is incident on a non-SP edge. In the following, let (u, v) be v's only incoming edge in G_{SP}. This is the most complicated case because a fork can become an anchor node and anchor predecessors and successors for other nodes can change. We have two cases depending on whether a fork becomes an anchor. Following both cases, we deal with the impact of making v an anchor on other nodes.

Case 3a: $\mathit{PredAnchor}(u)$ has no outgoing edges in $\mathcal{R}_{\text{SP}}$. In our example, this case occurs, e.g., when processing node 9. (figure 8.2 shows $\mathcal{R}$ after processing 9.)

In this case, no fork is made into an anchor. Simply add v and the edge $(\mathit{PredAnchor}(u), v)$ to $\mathcal{R}_{\text{SP}}$, and set $\mathit{PredAnchor}(v) = v$ and $\mathit{proxy}(v) = v$.

Case 3b: $\mathit{PredAnchor}(u)$ has an outgoing edge in $\mathcal{R}_{\text{SP}}$. In our example (see figure 8.1), this case occurs when processing node 24.

First, identify the fork node by performing a backwards graph search from v in G_{SP} until reaching a node f with $\mathit{proxy}(f) \neq \text{NULL}$.

Second, add f and v to $\mathcal{R}$ as follows. For the subsequent step, it will be convenient to reference certain old values, so temporarily store $p_f = \mathit{PredAnchor}(f)$ and $s_f = \text{GetSuccAnchor}(f)$.

[3]For all of these updated nodes, the anchor successor will never change again. We could explicitly store $\mathit{SuccAnchor}(x) = j$ in these cases. But since we need the proxy for other situations, we use it here as well.

Add f and v to $\mathcal{R}_{\text{SP}}$; replace the edge (f, v) by edges (p_f, f) and (f, s_f); add the edge (f, v).[4] The nodes f and v are now anchors, so set $\mathit{PredAnchor}(f) = f$, $\mathit{proxy}(f) = f$, $\mathit{PredAnchor}(v) = v$, and $\mathit{proxy}(v) = v$.

Next, update any nodes whose anchor predecessor and proxy should change. Specifically, perform a graph search forward from f in G_{SP}, truncating the search whenever reaching nodes x with $\mathit{PredAnchor}(x) \neq p_f$. Consider each node x reached during the search with $\mathit{PredAnchor}(x) = p_f$. For each of these nodes, update $\mathit{PredAnchor}(x) = f$. Also for each of these nodes, if $\mathit{proxy}(x) = p_f$, update it to $\mathit{proxy}(x) = f$.

In both cases: Add the non-SP edge to the reachability structure for $\mathcal{R}$. Perform a backwards graph search in G_{SP} from u, only visiting those nodes x for which $\mathit{proxy}(x) = \text{NULL}$. Set $\mathit{proxy}(x) = \mathit{PredAnchor}(u)$ for those nodes.

8.4 Correctness Proof

The main correctness argument boils down to showing that the values maintained by the algorithm match the definitions. We have already shown that the defined structures are sufficient to support queries, so we get overall correctness as a corollary. The proof is tedious, with cases matching each case of the algorithm, but there is nothing surprising therein.

Lemma 8.10. *The algorithm correctly maintains anchors, anchor predecessors, proxies, and the graph $\mathcal{R}_{\text{SP}}$ according to the definitions specified in sections 8.1 and 8.2.*

Proof. By induction over processing nodes, with each case of the algorithm considered separately. Note that a non-anchor node has no impact on $\mathcal{R}$, the anchor predecessor, or the

[4] We do not have to remove the redundant edge (p_f, s_f) from the non-SP graph $\mathcal{R}$ as it does not change the transitive closure.

anchor successor of other nodes. Moreover, a node being processed has no successors. Thus, Case 1 trivially does what it needs to—update the info for the node v itself.

(*Case 2.*) The join j should only become an anchor if both subdags have anchors. If both subdags have an anchor, then those anchors cannot be related to each other. Thus, $p_j = j$ if and only if both subdags have anchors.

If $p_j \neq j$, then j correctly chooses its predecessor by inheriting the nearer anchor predecessor from its two incoming edges. Nothing else changes.

If $p_j = j$, then j indeed becomes an anchor, and the edges added to $\mathcal{R}_{\text{SP}}$ are exactly as defined. Since j has no successors, no other node's predecessor should change. But j may be the anchor successor of some nodes—specifically, only predecessors with no current anchor successors. If a node has an anchor successor, then so do all its predecessors, so the search can truncate at nodes with defined proxies. Consider just the search on the left subdag (the other being symmetric). For each node reached x, the backward path from j to x either lies along a path to $\mathit{PredAnchor}(\ell)$ or not. Note that no diverging path can have an anchor by definition of anchor predecessor. Thus, if x is on the path to the predecessor, x should have $\mathit{proxy}(x) = \mathit{PredAnchor}(x) = \mathit{PredAnchor}(\ell)$. If x is not on the path, then the path to x must follow a different branch from a join j', and x's branch cannot have any anchors, so we should have $\mathit{proxy}(x) = \mathit{PredAnchor}(j') = \mathit{PredAnchor}(\ell)$. Either way, the proxy is set correctly.

(*Claim: if a node has* $\mathit{proxy}(x)$ *set to a non-predecessor (i.e., the join case), then its proxy should never change again.*) Let j' be the join such that $\mathit{proxy}(x) = \mathit{PredAnchor}(j')$. All predecessors of j' have already executed, so x's situation with respect to j' cannot change.

The implication is that we only need to worry about updating the proxy again if it satisfies the other situations. Most notably, if $\mathit{proxy}(x) = \mathit{PredAnchor}(x)$, then $\mathit{proxy}(x)$ should change when $\mathit{PredAnchor}(x)$ changes.

(*Case 3.*) We first argue that case 3a and 3b correctly identify when a fork should become an anchor. If $\mathit{PredAnchor}(v)$ does not have any outgoing edges in $\mathcal{R}_{\text{SP}}$, then there is no fork between $\mathit{PredAnchor}(v)$ and v with any anchor successor. If $\mathit{PredAnchor}(v)$ does have such an edge, then the path to the target of the edge must diverge at some fork f from the path to v. The backwards search finds the point of divergence: the nearest predecessor to v with an anchor successor. Thus, cases 3b correctly anchorizes fork nodes. In both cases 3a and 3b, the edges added to $\mathcal{R}$ correspond exactly to the definition.

Adding f as an anchor only changes the anchor predecessor for those nodes that formerly had $\mathit{PredAnchor}(f)$ as their anchor predecessor. Since these must all be connected, the forward search corrects these. Moreover, according to the claim, the proxy should continue to track the anchor predecessor. No proxy or anchor predecessor for any node preceding f should change.

Making v itself an anchor is similar to (but slightly simpler than) case 2. □

Corollary 8.11. *For any two already-processed nodes $u, v \in V$, the* QUERY(u, v) *correctly returns* TRUE *if $u \prec_G v$ and* FALSE *otherwise.*

Proof. By lemma 8.4, we just need to prove properties 8.1 and 8.3. These are implied by lemmas 8.7 to 8.9 as long as the algorithm satisfies lemma 8.10. □

8.5 Performance Analysis

This section argues that the construction algorithm runs in total time $O(n + k^2)$ when using SP-Order for G_{SP} and dynamically computing the transitive closure of $\mathcal{R}$, where $n = |V|$ is the number of vertices and $k = |E_{\text{non}}|$ is the number of non-SP edges.

The most worrisome part of the algorithm is that it performs graph searches which may, in the worst case, traverse the entire graph. Here we provide some lemmas that charge these searches against certain changes that can only occur a constant number of times.

The following lemma charges the cost of a backwards search against changing the node's proxy from NULL, which can only happen once. The key observation for Case 3 is that both backwards searches touch the same nodes.

Lemma 8.12. *For each node visited by a backward graph search, there are* $\Omega(1)$ *nodes whose proxy changes from* NULL *to something else.*

Proof. The backwards graph searches occur in Case 2 and Case 3. The former is easier as the search directly changes the proxy. The only nodes visited are those with $\mathit{proxy}(x) = \text{NULL}$ (and their neighbors to decide when to stop). Since the series-parallel graph has in-degree zero, this is $\Omega(1)$ nodes per change.

Case 3 performs up to two backwards search. The first does not directly change any proxies. However, it still only traverses nodes with a NULL proxy. These nodes will all be visited again in the second graph search, which matches the case. The cost here thus increases by a constant factor. □

Bounding the cost of forward searches is less obvious. We state the two key helper lemmas here, followed by the main result: each node can only be involved in two forward searches, once when it is in the right subdag of a fork that becomes an anchor, and once when it is in the left subdag of a fork that becomes an anchor. An important assumption for these proofs is the depth-first execution order.

Lemma 8.13. *Suppose the graph is processed in the specified depth-first order. Let f be a fork and let f_ℓ be a fork nested in f's left subdag. Suppose that both forks eventually become anchors. Then either the nested fork f_ℓ becomes an anchor before f does, or f becomes an anchor before processing the node f_ℓ.*

Proof. Suppose first we have not yet started processing f's right subdag at the point that f_ℓ is processed. Then by depth-first order, we cannot process any of f's right subdag until both of f_ℓ's subdags execute to the point they either complete (which means processing anchors by assumption) or block (also processing an anchor). So f_ℓ becomes an anchor before f_r.

Suppose instead that f's right subdag has started processing. Then at least one anchor must have already been processed in f's left subdag. The execution only jumps out of f's right subdag if a principle anchor is processed, either because that principle anchor is blocked or because some other node becomes unblocked. In either case, f has anchors in both subdags, so f becomes an anchor. □

Lemma 8.14. *Suppose the graph is processed in the specified depth-first order. Let f be a fork and let f_r be a fork nested in f's right subdag. Suppose that both forks eventually become anchors. Then f becomes an anchor before processing any of f_r's right subdag.*

Proof. The right subdag of f only starts executing when the left subdag has completed (having processed an anchor) or is blocked on anchors. In either case, by the time the first

principle anchor is processed in f_r, f becomes an anchor. That would be while still processing f_r's left subtree. □

Lemma 8.15. *If processing nodes in the specified depth-first order, each node can be visited by at most 2 forward searches.*

Proof. Consider a particular node x. We claim that the two times x can be visited by forward searches are: (1) the first time that x is in the right subdag of a fork that becomes an anchor, and (2) the first time that x is in the left subdag of a fork that becomes an anchor.

By inspection, forwards searches only occur when forks become anchors. Moreover, there must be a new anchor in one of the fork's subdags to trigger the anchoring. Thus the corresponding join cannot have executed yet, and we need not worry about the forward search exiting the fork's subdags.

(Right subdag.) Suppose for the sake of contradiction that a particular node x is visited by two forward searches, from f and f_r, while in the right subdag of both forks. Without loss of generality, let f_r be in the right subdag of f. (Parallel composition has to nest.) Then by lemma 8.14, f becomes an anchor before x is processed. This contradicts the assumption that x was involved in both searches.

(Left subdag.) Suppose for the sake of contradiction that a particular node x is visited by two forward searches, from f and f_ℓ, while in the left subdag of both forks. Without loss of generality, let f_ℓ be in the left subdag of f. Then by lemma 8.13, we have two options. If f_ℓ becomes an anchor before f, then the forwards search from f would not pass through f, contradicting the assumption that both searches reach x. If not, f becomes an anchor before processing f_ℓ and hence before processing x, and the forward search from f cannot touch x. □

We now give our main performance theorem. Substituting in the time required to build the transitive closure, we get a total construction time of $O(n + k^2)$ and query time of $O(1)$.

Theorem 8.16. *Let $n = |V|$ be the number of vertices and let $k = |E_{\text{non}}|$ be the number of non-SP edges in the input graph G.*

Consider any incremental data structure that supports reachability queries on general graphs, supporting both edge insertions and queries. Let I be the total time to perform $\Theta(k)$ edge insertions, and let Q be the the time per query.

Then our construction algorithm runs in a total of $O(n+I)$ time, and it answers reachability queries on any two already-processed nodes in $O(Q)$ time.

Proof. Almost all of the steps in the algorithm to process each node is clearly constant time (e.g., looking at pointers, updating pointers, etc.). The exceptions are the following: (1) a constant number of insertions into our reachability structure for G_{SP}, (2) a constant number of insertions into our reachability structure for $\mathcal{R}$, (3) a constant number of reachability queries on G_{SP}, (4) the call to GetSuccAnchor, which is dominated by a reachability query on G_{SP}, and (5) the graph searches. Using an efficient data structure for the base SP graph (say, SP-Order), all steps except the graph searches and the insertions into $\mathcal{R}$ require constant time per node processed.

The total number of edges in $\mathcal{R}$ is a constant per node anchor node for the edges in $\mathcal{R}_{\text{SP}}$, plus k for the non-SP edges. By lemma 8.5, the total number of edges is thus $O(k)$, and hence the total cost of construction is $O(I)$.

To bound the backward searches in aggregate, we apply lemma 8.12. Each only changes from *proxy* = NULL once over the course of the algorithm, so we can charge the searches to those changes. The total cost of backward searches is thus $O(n)$.

We bound forward searches using lemma 8.15, which says that each node can only be visited twice. Again we have a total cost of $O(n)$. □

8.5.1 Full Race Detection

We have shown the efficiency of the dynamic reachability algorithm. This can be extended for a full race detection algorithm by adding an access history component. Note, again, that unlike series-parallel computations, each write may generate multiple queries; however, for the same reason as explained in section 7.3 the total number of queries is bounded by two times the total number of reads since each writer removes the entire reader-list. Therefore, the total running time is for race detection is $O(T_1 + k^2)$.

8.6 Related Work

Related to, but easier than, our problem is the static problem of building a reachability oracle for a directed graph. In the case where the base graph is an n-node directed tree and k non-tree edges are added, Wang et al. [187] provide an algorithm that requires $O(n + k^2)$ space, $O(n+k^3)$ construction time, and answers queries in $O(1)$ time. Our algorithm achieves the same space, a better construction time, and works for more general series-parallel graphs. Moreover, our algorithm is incremental, supporting edge insertions, whereas their algorithm is offline. Also related is the problem of labeling each vertex (offline) such that queries can be answered by simply comparing vertex labels. The best practical algorithm we are aware

of uses 2-hop labels [49], but its construction time is polynomial, and there is no nontrivial bound known for the label size and hence query time once arbitrary edges are included in the graph.

Related work for race detection for fork/join programs is discussed in section 4.5. Race detection for other models is mentioned in section 7.4. To our knowledge there exists only one algorithm exists for determinacy race detection for dynamic multithreading with futures. Surendran and Sarkar [174] presented an online, serial algorithm for this problem. Given a program that normally executes in T_1 time on a single processor, that algorithm executes in time

$$O\left(T(k+1)(n+1)\alpha(T, s+k)\right)$$

and space

$$O\left(s+k+n+v(k+1)\right)$$

where k is the number of futures created, s is the number of spawned tasks, n is the number of future touches, v is the number of shared memory locations, and α is the inverse Ackermann function.

8.7 Conclusions and Future Work

This chapter presented a race-detection algorithm for general future-parallel programs that runs in time $O(T_1 + k^2)$. By handling arbitrary futures, our system automatically handles

pipeline parallelism, a weaker but very common expression of parallelism. It is an open question whether a more specialized technique exists for race detection in pipeline parallel DAGs which may be more efficient.

Note that if one is not careful, a program with unstructured futures can deadlock. Such a deadlock is deterministic, however, and does not depend on the schedule. In such cases, our algorithm detects races until the execution deadlocks.

The most glaring opportunity for future work is an implementation of this algorithm, comparing it to Surendran and Sarkar's [174] algorithm. The algorithm presented here would be rather complicated to implement, requiring that we maintain the entire computation graph is memory in order for the graph searches. It would be interesting to consider similar algorithms that may be able to avoid the graph searches. There may also be different algorithms which reduce the overhead of the k^2 term.

In addition, our current algorithm executes a computation serially. In future work we hope to develop a parallel algorithm for race detection, either as an extension to the presented algorithm or via a new algorithm.

Chapter 9

Conclusions and Future Directions

It is increasing important to write software that utilizes modern multicore hardware. To offset the new challenges that parallel programming brings, we will need different techniques and tools for programming such hardware. Concurrency platforms like Cilk Plus [99] have decreased the difficulty in parallel programming, but more can be done. This dissertation has explored how runtime systems can be designed to support better tools to improve parallel programming.

Specifically, we have explored the design space around allowing Cilk workers to have multiple deques, considering the consequences of each design point to dynamic tools to ease parallel programming. The result is a set of runtime systems and tools which make parallel programming a more pleasant experience.

Chapter 3 presented Batcher, a runtime scheduler that automatically groups concurrent data structure operations into groups and invokes operations on a batched data structure,

efficiently scheduling work on this data structure and on the core computation. The result allows batched data structures to be treated essentially as sequential data structure operations, including a theoretical performance guarantee.

A specialization of that runtime scheduler is provided in chapter 4, yielding a tool that automatically detects determinacy races in Cilk programs. By leveraging custom runtime support, CRacer runs efficiently and in parallel, reducing the time for the typical develop-test-debug feedback loop.

Chapter 5 covers an extension to the Cilk Plus runtime system that allows non-blocking suspension and resumption at arbitrary points. This can be used to support more advanced synchronization than is available in Cilk (fork/join programs) as well as hiding the latency of external events, such as network operations or I/O.

PORRidge, a debugging tool for non-deterministic parallel programs, is designed and examined in chapter 6. While existing record-and-replay tools target persistent thread models, PORRidge leverages the structure of fork/join dynamically multithreaded programs to provide process-oblivious record and replay of lock acquisitions in Cilk programs. PORRidge comes with a nearly-optimal performance bound and has been shown to perform well in practice.

Finally, chapters 7 and 8 extends the work in chapter 4 with the goal of detecting races for broader classes of computations. In particular, it considers the of **futures**, which can be used to generate arbitrary computation DAGs. A useful but restricted use of futures is considered in chapter 7, while chapter 8 handles any use of futures. In both cases we design efficient race detection systems for such computations. Work on implementing these systems,

using the runtime from chapter 5 to bring futures into the purview of Cilk's work stealing scheduler, is in-progress.

In fact, none of these runtime systems contain mutually exclusive features. With some engineering effort, these could be merged into a single runtime system that may serve as an important platform for future research.

9.1 Future Work

Of course, the investigations in this dissertation represent only a sample of this research space. As Matt Might describes [133], a dissertation is just a small dent in the boundaries of human knowledge. In addition to the many ways each individual project may be improved (engineering improvements, improved performance bounds, etc.), there are many directions not addressed by this dissertation.

Though we have mostly considered how runtime support can improve tools for parallel programming, there are many levels in the software development stack that might be reconsidered in the context of parallel runtime systems. For example, Lee et al. [120] explore how an extension to Linux's virtual memory system can be used to improve the efficiency and interoperability of the Cilk Plus runtime system.

This dissertation has focused only on shared-memory homogeneous systems. With the popularity of multicore architectures we have also seen fresh exploration of **heterogeneous** hardware designs, where different cores may have different performance characteristics. These parallel machines can be particularly difficult to program, but some of the burden might be

shifted to the runtime system if they can hide this heterogeneity and provide good performance. The same can be said for distributed systems, where communication between subsystems must be an integral consideration of any scheduler.

Perhaps the largest gap in our understanding of this area pertains to **locality** of memory use. Even in the sequential systems there is a large gap between memory latency and clock speeds, leading to decreasing processor utilization over the years. Memory latency is often worse in parallel machines, where some memory may be closer to some cores than others and thus transfer times may differ depending on location. Runtime systems may find increasing use in automatically handling some memory transfers, reducing complexity in much the same way that runtime garbage collectors reduce the complexity of manual memory management. Further, though there is plenty of work on minimizing cache misses for sequential algorithms (e.g. [18, 78], understanding cache misses in the context of dynamic scheduling is a harder problem [92, 44, 2]. In particular, it is unclear how a runtime scheduler might strike the right balance between load balance and locality of reference.

Bibliography

[1] International Standard ISO/IEC 14882:2011. *Programming Languages – C++*. International Organization for Standards, 2011.

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. "The Data Locality of Work Stealing". In: *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '00. Bar Harbor, Maine, USA: ACM, 2000, pp. 1–12. ISBN: 1-58113-185-2. DOI: 10.1145/341800.341801. URL: http://doi.acm.org/10.1145/341800.341801.

[3] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. "Helper Locks for Fork-join Parallel Programming". In: *SIGPLAN Not.* 45.5 (Jan. 2010), pp. 245–256. ISSN: 0362-1340. DOI: 10.1145/1837853.1693487. URL: http://doi.acm.org/10.1145/1837853.1693487.

[4] Kunal Agrawal et al. "Provably Good Scheduling for Parallel Programs That Use Data Structures Through Implicit Batching". In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '14. Prague, Czech Republic: ACM, 2014, pp. 84–95. ISBN: 978-1-4503-2821-0. DOI: 10.1145/2612669.2612688. URL: http://doi.acm.org/10.1145/2612669.2612688.

[5] Todd R. Allen and David A. Padua. "Debugging Fortran on a Shared Memory Machine". In: *Proc. of the 1987 International Conference on Parallel Processing*. ICPP '87. Aug. 1987, pp. 721–727.

[6] Gautam Altekar and Ion Stoica. "ODR: Output-Deterministic Replay for Multicore Debugging". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 193–206. ISBN: 978-1-60558-752-3. DOI: http://doi.acm.org/10.1145/1629575.1629594.

[7] Arne Andersson. "Improving Partial Rebuilding by Using Simple Balance Criteria". In: *Proceedings of the Workshop on Algorithms and Data Structures*. WADS '89. London, UK, UK: Springer-Verlag, 1989, pp. 393–402. ISBN: 3-540-51542-9. URL: http://dl.acm.org/citation.cfm?id=645928.672546.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. “Thread Scheduling for Multiprogrammed Multiprocessors”. In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’98. Puerto Vallarta, Mexico: ACM, 1998, pp. 119–129. ISBN: 0-89791-989-0. DOI: 10.1145/277651.277678. URL: http://doi.acm.org/10.1145/277651.277678.

[9] Arvind, R.S. Nikhil, and K.K. Pingali. “I-structures: Data Structures for Parallel Computing”. In: *Proceedings of the Graph Reduction Workshop*. 1986.

[10] Amittai Aviram et al. “Efficient System-Enforced Deterministic Parallelism”. In: Vancouver, BC, Canada, 2010, pp. 1–16.

[11] B. Awerbuch et al. “Minimizing the Flow Time without Migration”. In: *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*. May 1999, pp. 198–205.

[12] E. Ayguade et al. “The Design of OpenMP Tasks”. In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (Mar. 2009), pp. 404–418. URL: http://dx.doi.org/10.1109/TPDS.2008.105.

[13] Yossi Azar et al. “Balanced Allocations”. In: *SIAM Journal Comput.* 29.1 (Sept. 1999), pp. 180–200. ISSN: 0097-5397. DOI: 10.1137/S0097539795288490. URL: http://dx.doi.org/10.1137/S0097539795288490.

[14] Henry C. Baker Jr. and Carl Hewitt. “The Incremental Garbage Collection of Processes”. In: *SIGPLAN Not.* 12.8 (Aug. 1977), pp. 55–59. ISSN: 0362-1340. DOI: 10.1145/872734.806932. URL: http://doi.acm.org/10.1145/872734.806932.

[15] Rajkishore Barik et al. “The Habanero Multicore Software Research Project”. In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009, pp. 735–736. ISBN: 978-1-60558-768-4.

[16] R. Bayer and M. Schkolnick. “Concurency of Operations on B-Trees”. In: *Acta Informatica* 9 (1977), pp. 1–21.

[17] Rob von Behren et al. “Capriccio: Scalable Threads for Internet Services”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: ACM, 2003, pp. 268–281. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945471. URL: http://doi.acm.org/10.1145/945445.945471.

[18] Michael A. Bender et al. “Concurrent Cache-Oblivious B-Trees”. In: *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Las Vegas, NV, USA, July 2005, pp. 228–237.

[19] Michael A. Bender et al. “On-the-fly Maintenance of Series-parallel Relationships in Fork-join Multithreaded Programs”. In: *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’04. Barcelona, Spain: ACM, 2004, pp. 133–144. ISBN: 1-58113-840-7. DOI: 10.1145/1007912.1007933. URL: http://doi.acm.org/10.1145/1007912.1007933.

[20] Michael A. Bender et al. “Two Simplified Algorithms for Maintaining Order in a List”. In: *Proceedings of the 10th Annual European Symposium on Algorithms*. ESA '02. London, UK, UK: Springer-Verlag, 2002, pp. 152–164. ISBN: 3-540-44180-8. URL: http://dl.acm.org/citation.cfm?id=647912.740822.

[21] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. “The Natural Work-Stealing Algorithm is Stable”. In: *SIAM J. Comput.* 32.5 (May 2003), pp. 1260–1279. ISSN: 0097-5397. DOI: 10.1137/S0097539701399551. URL: http://dx.doi.org/10.1137/S0097539701399551.

[22] Tom Bergan et al. “CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution”. In: *ASPLOS*. Pittsburgh, Pennsylvania, USA, 2010, pp. 53–64. DOI: http://doi.acm.org/10.1145/1736020.1736029.

[23] Tom Bergan et al. “Deterministic Process Groups in dOS”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 177–191. URL: http://dl.acm.org/citation.cfm?id=1924943.1924956.

[24] Emery D. Berger et al. “Grace: Safe Multithreaded Programming for C/C++”. In: Orlando, Florida, USA, 2009, pp. 81–96. DOI: http://doi.acm.org/10.1145/1640089.1640096.

[25] Christian Bienia and Kai Li. “Characteristics of Workloads Using the Pipeline Programming Model”. In: *Proceedings of the 3rd Workshop on Emerging Applications and Many-core Architecture*. June 2010.

[26] Christian Bienia et al. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. Oct. 2008.

[27] Guy E. Blelloch et al. “Internally deterministic parallel algorithms can be fast”. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. 2012, pp. 181–192. DOI: 10.1145/2145816.2145840. URL: http://doi.acm.org/10.1145/2145816.2145840.

[28] Robert D. Blumofe. “Executing Multithreaded Programs Efficiently”. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677. PhD thesis. Cambridge, Massachusetts: Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.

[29] Robert D. Blumofe and Charles E. Leiserson. “Scheduling multithreaded computations by work stealing”. In: *Journal of the ACM* 46 (1999).

[30] Robert D. Blumofe and Charles E. Leiserson. “Space-efficient Scheduling of Multithreaded Computations”. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*. STOC ’93. San Diego, California, USA: ACM, 1993, pp. 362–371. ISBN: 0-89791-591-7. DOI: 10.1145/167088.167196. URL: http://doi.acm.org/10.1145/167088.167196.

[31] Robert D. Blumofe et al. “Cilk: An Efficient Multithreaded Runtime System”. In: *Journal of Parallel and Distributed Computing* 37.1 (1996), pp. 55–69.

[32] Robert L. Bocchino Jr. et al. “Parallel Programming Must Be Deterministic by Default”. In: Berkeley, California, 2009, pp. 4–9.

[33] Michael D. Bond et al. “Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine”. In: 2015, pp. 90–101.

[34] Anastasia Braginsky and Erez Petrank. “A lock-free B+tree”. In: *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Pittsburgh, Pennsylvania, USA, 2012, pp. 58–67. ISBN: 978-1-4503-1213-4. DOI: 10.1145/2312005.2312016. URL: http://doi.acm.org/10.1145/2312005.2312016.

[35] Richard P. Brent. “The Parallel Evaluation of General Arithmetic Expressions”. In: *J. ACM* 21.2 (Apr. 1974), pp. 201–206. ISSN: 0004-5411. DOI: 10.1145/321812.321815. URL: http://doi.acm.org/10.1145/321812.321815.

[36] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. “A Parallel Priority Queue with Constant Time Operations”. In: *Journal of Parallel and Distributed Computing* (1998), pp. 4–21.

[37] Zoran Budimlić et al. “Concurrent Collections”. In: *Journal of Scientific Programming* 18.3-4 (Aug. 2010), pp. 203–217. ISSN: 1058-9244.

[38] Jan Bulánek, Michal Koucký, and Michael Saks. “On Randomized Online Labeling with Polynomially Many Labels”. English. In: *Automata, Languages, and Programming*. Vol. 7965. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 291–302. ISBN: 978-3-642-39205-4. DOI: 10.1007/978-3-642-39206-1_25. URL: http://dx.doi.org/10.1007/978-3-642-39206-1_25.

[39] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. “Concurrent programming with revisions and isolation types”. In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 2010, pp. 691–707. DOI: 10.1145/1869459.1869515. URL: http://doi.acm.org/10.1145/1869459.1869515.

[40] Vincent Cavé et al. “Habanero-Java: The New Adventures of Old X10”. In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ ’11. Kongens Lyngby, Denmark: ACM, 2011, pp. 51–61. ISBN: 978-1-4503-0935-6. DOI: 10.1145/2093157.2093165. URL: http://doi.acm.org/10.1145/2093157.2093165.

[41] B.L. Chamberlain, D. Callahan, and H.P. Zima. “Parallel Programmability and the Chapel Language”. In: *Int. J. High Perform. Comput. Appl.* 21.3 (Aug. 2007), pp. 291–312. ISSN: 1094-3420. DOI: 10.1177/1094342007078442. URL: http://dx.doi.org/10.1177/1094342007078442.

[42] Rohit Chandra, Anoop Gupta, and John L. Hennessy. “COOL: An Object-Based Language for Parallel Programming”. In: *IEEE Computer* 27.8 (Aug. 1994), pp. 13–26.

[43] Philippe Charles et al. “X10: An Object-oriented Approach to Non-uniform Cluster Computing”. In: *SIGPLAN Not.* 40.10 (Oct. 2005), pp. 519–538. ISSN: 0362-1340. DOI: 10.1145/1103845.1094852. URL: http://doi.acm.org/10.1145/1103845.1094852.

[44] Shimin Chen et al. “Scheduling Threads for Constructive Cache Sharing on CMPs”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’07. San Diego, California, USA: ACM, 2007, pp. 105–115. ISBN: 978-1-59593-667-7. DOI: 10.1145/1248377.1248396. URL: http://doi.acm.org/10.1145/1248377.1248396.

[45] Guang-Ien Cheng et al. “Detecting data races in Cilk programs that use locks”. In: *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’98. 1998.

[46] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. “Techniques for Debugging Parallel Programs with Flowback Analysis”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 491–530. ISSN: 0164-0925. DOI: 10.1145/115372.115324. URL: http://doi.acm.org/10.1145/115372.115324.

[47] Jong-Deok Choi et al. “Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI ’02. Berlin, Germany: ACM, 2002, pp. 258–269.

[48] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. “A Universal Construction for Wait-free Transaction Friendly Data Structures”. In: *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’10. Thira, Santorini, Greece: ACM, 2010, pp. 335–344. ISBN: 978-1-4503-0079-7. DOI: 10.1145/1810479.1810538. URL: http://doi.acm.org/10.1145/1810479.1810538.

[49] Edith Cohen et al. “Reachability and Distance Queries via 2-hop Labels”. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 937–946. ISBN: 0-89871-513-X. URL: http://dl.acm.org/citation.cfm?id=545381.545503.

[50] Richard Cole and Vijaya Ramachandran. “Resource oblivious sorting on multicores”. In: *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*. Bordeaux, France, 2010, pp. 226–237. ISBN: 3-642-14164-1, 978-3-642-14164-5. URL: http://dl.acm.org/citation.cfm?id=1880918.1880944.

[51] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[52] Intel Corporation. *Automated Relational Knowledge Base (Intel ARK)*. URL: https://ark.intel.com (visited on 06/21/2017).

[53] A. Crauser et al. “A Parallelization of Dijkstra’s Shortest Path Algorithm”. In: *Proceedings of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*. Springer, 1998, pp. 722–731.

[54] Joseph Devietti et al. “DMP: Deterministic Shared Memory Multiprocessing”. In: *ASPLOS*. Washington, DC, USA, 2009, pp. 85–96. DOI: http://doi.acm.org/10.1145/1508244.1508255.

[55] Joseph Devietti et al. “RADISH: Always-on Sound and Complete Race Detection in Software and Hardware”. In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA ’12. Portland, Oregon: IEEE Computer Society, 2012, pp. 201–212.

[56] Joseph Devietti et al. “RCDC: A Relaxed Consistency Deterministic Computer”. In: *ASPLOS*. Newport Beach, California, USA, 2011, pp. 67–78. DOI: http://doi.acm.org/10.1145/1950365.1950376.

[57] P. Dietz and D. Sleator. “Two Algorithms for Maintaining Order in a List”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC ’87. New York, New York, USA: ACM, 1987, pp. 365–372. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28434. URL: http://doi.acm.org/10.1145/28395.28434.

[58] Paul F. Dietz. “Maintaining Order in a Linked List”. In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. STOC ’82. San Francisco, California, USA: ACM, 1982, pp. 122–127. ISBN: 0-89791-070-2. DOI: 10.1145/800070.802184. URL: http://doi.acm.org/10.1145/800070.802184.

[59] E. W. Dijkstra. “Co-operating Sequential Processes”. In: *Programming Languages*. Ed. by F. Genuys. Originally published as Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965. London, England: Academic Press, 1968, pp. 43–112.

[60] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. "Race Detection in Two Dimensions". In: *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '15. Portland, Oregon, USA: ACM, 2015, pp. 101–110. ISBN: 978-1-4503-3588-1. DOI: 10.1145/2755573.2755601. URL: http://doi.acm.org/10.1145/2755573.2755601.

[61] A. Dinning and E. Schonberg. "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection". In: *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*. PPOPP '90. Seattle, Washington, USA: ACM, 1990, pp. 1–10. ISBN: 0-89791-350-7. DOI: 10.1145/99163.99165. URL: http://doi.acm.org/10.1145/99163.99165.

[62] James R. Driscoll et al. "Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation". In: *Communications of the ACM* 31 (11 1988), pp. 1343–1354. ISSN: 0001-0782.

[63] George W. Dunlap et al. "Execution Replay of Multiprocessor Virtual Machines". In: Seattle, WA, USA, 2008, pp. 121–130. DOI: http://doi.acm.org/10.1145/1346256.1346273.

[64] D. L. Eager, J. Zahorjan, and E. D. Lozowska. "Speedup Versus Efficiency in Parallel Systems". In: *IEEE Trans. Comput.* 38.3 (Mar. 1989), pp. 408–423. ISSN: 0018-9340. DOI: 10.1109/12.21127. URL: http://dx.doi.org/10.1109/12.21127.

[65] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. "Goldilocks: A Race and Transaction-aware Java Runtime". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 245–255. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250762. URL: http://doi.acm.org/10.1145/1250734.1250762.

[66] Perry A. Emrath and David A. Padua. "Automatic Detection of Nondeterminacy in Parallel Programs". In: *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*. PADD '88. Madison, Wisconsin, USA: ACM, 1988, pp. 89–99. ISBN: 0-89791-296-9. DOI: 10.1145/68210.69224. URL: http://doi.acm.org/10.1145/68210.69224.

[67] Dawson Engler and Ken Ashcraft. "RacerX: Effective, Static Detection of Race Conditions and Deadlocks". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 237–252. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945468. URL: http://doi.acm.org/10.1145/945445.945468.

[68] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. "Parallel Bi-objective Shortest Paths Using Weight-Balanced B-trees with Bulk Updates". In: *Proceedings of the 13th International Symposium on Experimental Algorithms - Volume 8504*. New York,

NY, USA: Springer-Verlag New York, Inc., 2014, pp. 111–122. ISBN: 978-3-319-07958-5. DOI: 10.1007/978-3-319-07959-2_10. URL: http://dx.doi.org/10.1007/978-3-319-07959-2_10.

[69] John Erickson et al. "Effective data-race detection for the kernel". In: *In Proceedings of the 9th USENIX conference on Operating systems design and implementation*. 2010.

[70] Panagiota Fatourou and Nikolaos D. Kallimanis. "A Highly-efficient Wait-free Universal Construction". In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011, pp. 325–334. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989549. URL: http://doi.acm.org/10.1145/1989493.1989549.

[71] Mingdong Feng and Charles E. Leiserson. "Efficient Detection of Determinacy Races in Cilk Programs". In: *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '97. Newport, Rhode Island, USA: ACM, 1997, pp. 1–11. ISBN: 0-89791-890-8. DOI: 10.1145/258492.258493. URL: http://doi.acm.org/10.1145/258492.258493.

[72] Cormac Flanagan and Stephen N. Freund. "FastTrack: Efficient and Precise Dynamic Race Detection". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 121–133. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542490. URL: http://doi.acm.org/10.1145/1542476.1542490.

[73] Matthew Fluet et al. "Implicitly Threaded Parallelism in Manticore". In: *Journal of Functional Programming* 20.5-6 (Nov. 2010), pp. 537–576. ISSN: 0956-7968. DOI: 10.1017/S0956796810000201. URL: http://dx.doi.org/10.1017/S0956796810000201.

[74] Michael L. Fredman and Dan E. Willard. "Surpassing the Information Theoretic Bound with Fusion Trees". In: *J. Comput. Syst. Sci.* 47.3 (Dec. 1993), pp. 424–436. ISSN: 0022-0000. DOI: 10.1016/0022-0000(93)90040-4. URL: http://dx.doi.org/10.1016/0022-0000(93)90040-4.

[75] Leonor Frias and Johannes Singler. "Parallelization of Bulk Operations for STL Dictionaries". In: *Euro-Par Workshops*. Vol. 4854. LNCS. Springer, 2007, pp. 49–58.

[76] D. P. Friedman and D. S. Wise. "Aspects of Applicative Programming for Parallel Processing". In: *IEEE Trans. Comput.* 27.4 (Apr. 1978), pp. 289–296. ISSN: 0018-9340. DOI: 10.1109/TC.1978.1675100. URL: http://dx.doi.org/10.1109/TC.1978.1675100.

[77] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The Implementation of the Cilk-5 Multithreaded Language". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998, pp. 212–223. ISBN: 0-89791-987-4. DOI: 10.1145/277650.277725. URL: http://doi.acm.org/10.1145/277650.277725.

[78] Matteo Frigo et al. “Cache-Oblivious Algorithms”. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. FOCS ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 285–. ISBN: 0-7695-0409-4. URL: http://dl.acm.org/citation.cfm?id=795665.796479.

[79] Matteo Frigo et al. “Reducers and Other Cilk++ Hyperobjects”. In: *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 2009, pp. 79–90.

[80] A. Fu and T. Kameda. “Concurrency Control of Nested Transactions Accessing B-Trees”. In: *PODS*. 1989, pp. 270–285.

[81] Igal Galperin and Ronald L. Rivest. “Scapegoat Trees”. In: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’93. Austin, Texas, USA: Society for Industrial and Applied Mathematics, 1993, pp. 165–174. ISBN: 0-89871-313-7. URL: http://dl.acm.org/citation.cfm?id=313559.313676.

[82] A. Georges et al. “JaRec: A Portable Record/Replay Environment for Multi-threaded Java Applications”. In: 34.6 (2004), pp. 523–547. ISSN: 0038-0644. DOI: 10.1002/spe.579.

[83] James R. Goodman, Mary K. Vernon, and Philip J. Woest. “Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors”. In: *SIGARCH Comput. Archit. News* 17.2 (Apr. 1989), pp. 64–75. ISSN: 0163-5964. DOI: 10.1145/68182.68188. URL: http://doi.acm.org/10.1145/68182.68188.

[84] N. Goodman and D. Shasha. “Semantically-Based Concurrency Control for Search Structures”. In: *PODS*. Mar. 1985, pp. 8–19.

[85] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390069X, 9780133900699.

[86] Robert H. Halstead Jr. “MULTILISP: A Language for Concurrent Symbolic Computation”. In: *ACM Trans. Program. Lang. Syst.* 7.4 (Oct. 1985), pp. 501–538. ISSN: 0164-0925. DOI: 10.1145/4472.4478. URL: http://doi.acm.org/10.1145/4472.4478.

[87] Johnson M. Hart. *Windows System Programming (3rd Edition)*. Addison-Wesley Professional, 2004. ISBN: 0321256190.

[88] Danny Hendler et al. “Flat Combining and the Synchronization-parallelism Tradeoff”. In: *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’10. Thira, Santorini, Greece: ACM, 2010, pp. 355–364. ISBN: 978-1-4503-0079-7. DOI: 10.1145/1810479.1810540. URL: http://doi.acm.org/10.1145/1810479.1810540.

[89] Maurice Herlihy. “A Methodology for Implementing Highly Concurrent Data Objects”. In: *ACM Transactions on Programming Languages and Systems* 15 (1993), pp. 745–770.

[90] Maurice Herlihy. “Wait-free synchronization”. In: *ACM Trans. Program. Lang. Syst.* 13 (1 Jan. 1991), pp. 124–149.

[91] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972.

[92] Maurice Herlihy and Zhiyu Liu. "Well-structured Futures and Cache Locality". In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '14. Orlando, Florida, USA: ACM, 2014, pp. 155–166. ISBN: 978-1-4503-2656-8. DOI: 10.1145/2555243.2555257. URL: http://doi.acm.org/10.1145/2555243.2555257.

[93] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123705916, 9780123705914.

[94] Lorin Hochstein et al. "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers". In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 35–. ISBN: 1-59593-061-2. DOI: 10.1109/SC.2005.53. URL: http://dx.doi.org/10.1109/SC.2005.53.

[95] Todd Hoff. *Linus: The Whole "Parallel Computing Is The Future" Is a Bunch of Crock.* 2014. URL: https://web.archive.org/web/20150810195954/http://highscalability.com/blog/2014/12/31/linus-the-whole-parallel-computing-is-the-future-is-a-bunch.html (visited on 06/13/2017).

[96] Derek R. Hower et al. "Two Hardware-Based Approaches for Deterministic Multiprocessor Replay". In: 52 (6 2009), pp. 93–100. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/1516046.1516068.

[97] Jeff Huang, Charles Zhang, and Julian Dolby. "CLAP: Recording Local Executions to Reproduce Concurrency Failures". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, Washington, USA, 2013, pp. 141–152. DOI: 10.1145/2491956.2462167.

[98] Institute of Electrical and Electronic Engineers. *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*. IEEE Standard 1003.1, 1996 Edition.

[99] Intel Corporation. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1*. Document 324396-002US. Available from http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm. Intel Corporation. 2013.

[100] *Intel Cilk Plus Language Specification*. Document Number: 324396-001US. Available from http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf. Intel Corporation. 2010.

[101] Arch D. Robison. *Cilk Plus Solver for a Chess Puzzle or: How I Learned to Love Fast Rejection*. https://software.intel.com/en-us/articles/cilk-plus-solver-for-a-chess-puzzle-or-how-i-learned-to-love-rejection. Feb. 2013.

[102] Intel. *Intel VTune Performance Analyzer*. http://www.intel.com/cd/software/products/asmona/eng/vtune/239144.htm. 2008.

[103] Intel Corporation. *An Introduction to the Cilk Screen race detector*. Nov. 2013. URL: https://software.intel.com/en-us/articles/an-introduction-to-the-cilk-screen-race-detector (visited on 11/30/2015).

[104] Intel Corporation. *CilkPlus/LLVM*. http://cilkplus.github.io/. 2013.

[105] Weixing Ji, Li Lu, and Michael L. Scott. "TARDIS: Task-level access race detection by intersecting sets". In: *Workshop on Determinism and Correctness in Parallel Programming*. WoDet '13. Houston, TX, USA, 2013.

[106] Theodore Johnson. "The Performance of Concurrent Data Structure Algorithms". PhD thesis. New York University, 1994.

[107] Theodore Johnson and Dennis Shasha. "A Framework for the Performance Analysis of Concurrent B-Tree Algorithms". In: *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 1990, pp. 273–287.

[108] Theodore Johnson and Dennis Shasha. "The Performance of Current B-tree Algorithms". In: *ACM Trans. Database Syst.* 18.1 (1993), pp. 51–101.

[109] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. "Concurrency and recovery in generalized search trees". In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. 1997, pp. 62–72.

[110] David A. Kranz, Robert H. Halstead Jr., and Eric Mohr. "Mul-T: A High-Performance Parallel Lisp". In: *PLDI*. ACM, 1989, pp. 81–90.

[111] H. T. Kung and P. L. Lehman. "Concurrent Manipulation of Binary Search Trees". In: *ACM Trans. on Database Systems* 5.3 (1980), pp. 339–353.

[112] Y. S. Kwong and D. Wood. "Method for Concurrency in B-Trees". In: *IEEE Trans. on Software Engineering* SE-8.3 (1982), pp. 211–223.

[113] Oren Laadan, Nicolas Viennot, and Jason Nieh. "Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems". In: New York, New York, USA, 2010, pp. 155–166. ISBN: 978-1-4503-0038-4. DOI: 10.1145/1811039.1811057. URL: http://doi.acm.org/10.1145/1811039.1811057.

[114] Richard E. Ladner and Michael J. Fischer. "Parallel Prefix Computation". In: *J. ACM* 27.4 (Oct. 1980), pp. 831–838. ISSN: 0004-5411. DOI: 10.1145/322217.322232. URL: http://doi.acm.org/10.1145/322217.322232.

[115] Vladimir Lanin and Dennis Shasha. "Concurrent set manipulation without locking". In: *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 1988, pp. 211–220.

[116] T. J. LeBlanc and J. M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay". In: 36 (4 1987), pp. 471–482. ISSN: 0018-9340. DOI: http://dx.doi.org/10.1109/TC.1987.1676929.

[117] Dongyoon Lee et al. "Chimera: Hybrid Program Analysis for Determinism". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, 2012, pp. 463–474. DOI: 10.1145/2254064.2254119.

[118] Dongyoon Lee et al. "Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism". In: *ASPLOS*. Pittsburgh, Pennsylvania, USA, 2010, pp. 77–90. DOI: http://doi.acm.org/10.1145/1736020.1736031.

[119] I-Ting Angelina Lee and Tao B. Schardl. "Efficiently Detecting Races in Cilk Programs That Use Reducer Hyperobjects". In: *SPAA '15: Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*. SPAA '15. Portland, Oregon, USA: ACM, June 2015, pp. 111–122. ISBN: 978-1-4503-3588-1. URL: http://doi.acm.org/10.1145/2755573.2755599.

[120] I-Ting Angelina Lee et al. "Using Memory Mapping to Support Cactus Stacks in Work-stealing Runtime Systems". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: ACM, 2010, pp. 411–420. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854324. URL: http://doi.acm.org/10.1145/1854273.1854324.

[121] Daan Leijen and Judd Hall. "Optimize Managed Code For Multi-Core Machines". In: *MSDN Magazine* (2007). Available from http://msdn.microsoft.com/magazine/.

[122] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. "The Design of a Task Parallel Library". In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, 2009, pp. 227–242. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640106. URL: http://doi.acm.org/10.1145/1640089.1640106.

[123] 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 22nd ACM symposium on Parallelism in algorithms and architectures*. Thira, Santorini, Greece, 2010, pp. 303–314. ISBN: 978-1-4503-0079-7. DOI: 10.1145/1810479.1810534. URL: http://doi.acm.org/10.1145/1810479.1810534.

[124] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. "Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms". In: *PPoPP*. 2012.

[125] Peng Liu et al. “Light: Replay via Tightly Bounded Recording”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, USA, 2015, pp. 55–64. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2738001.

[126] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. “Dthreads: Efficient Deterministic Multithreading”. In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, 2011, pp. 327–336. DOI: http://doi.acm.org/10.1145/2043556.2043587.

[127] Li Lu, Weixing Ji, and Michael L. Scott. “Dynamic Enforcement of Determinism in a Parallel Scripting Language”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 519–529. ISBN: 978-1-4503-2784-8.

[128] Chi-Keung Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[129] Reinhard Lüling and Burkhard Monien. “A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance”. In: *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’93. Velen, Germany: ACM, 1993, pp. 164–172. ISBN: 0-89791-599-2. DOI: 10.1145/165231.165252. URL: http://doi.acm.org/10.1145/165231.165252.

[130] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Available:http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html. Corvallis, OR, USA: kernel.org, 2011.

[131] John Mellor-Crummey. “Compile-time Support for Efficient Data Race Detection in Shared-memory Parallel Programs”. In: *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*. PADD ’93. San Diego, California, USA: ACM, 1993, pp. 129–139. ISBN: 0-89791-633-6. DOI: 10.1145/174266.171370. URL: http://doi.acm.org/10.1145/174266.171370.

[132] John Mellor-Crummey. “On-the-fly Detection of Data Races for Programs with Nested Fork-join Parallelism”. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing ’91. Albuquerque, New Mexico, USA: ACM, 1991, pp. 24–33. ISBN: 0-89791-459-7. DOI: 10.1145/125826.125861. URL: http://doi.acm.org/10.1145/125826.125861.

[133] Matt Might. *The Illustrated Guide to a Ph.D.* Matthew Might, 2010. URL: http://matt.might.net/articles/phd-school-in-pictures/.

[134] Michael Mitzenmacher. “Analyses of Load Stealing Models Based on Differential Equations”. In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’98. Puerto Vallarta, Mexico: ACM, 1998, pp. 212–221. ISBN: 0-89791-989-0. DOI: 10.1145/277651.277687. URL: http://doi.acm.org/10.1145/277651.277687.

[135] Stefan K. Muller and Umut A. Acar. “Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’16. Pacific Grove, California, USA: ACM, 2016, pp. 71–82. ISBN: 978-1-4503-4210-0. DOI: 10.1145/2935764.2935793. URL: http://doi.acm.org/10.1145/2935764.2935793.

[136] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. “Recording Shared Memory Dependencies Using Strata”. In: *ASPLOS*. San Jose, California, USA, 2006, pp. 229–240. DOI: http://doi.acm.org/10.1145/1168857.1168886.

[137] Robert H. B. Netzer and Barton P. Miller. “What Are Race Conditions?: Some Issues and Formalizations”. In: *ACM Lett. Program. Lang. Syst.* 1.1 (Mar. 1992), pp. 74–88. ISSN: 1057-4514. DOI: 10.1145/130616.130623. URL: http://doi.acm.org/10.1145/130616.130623.

[138] Robert Netzer and Barton P. Miller. “Detecting Data Races in Parallel Program Executions”. In: *In Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*. MIT Press, 1989, pp. 109–129.

[139] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. “Deterministic galois: on-demand, portable and parameterless”. In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS ’14, Salt Lake City, UT, USA, March 1-5, 2014*. Salt Lake City, Utah, USA: ACM, 2014, pp. 499–512. DOI: 10.1145/2541940.2541964. URL: http://doi.acm.org/10.1145/2541940.2541964.

[140] Itzhak Nudler and Larry Rudolph. “Tools for the Efficient Development of Efficient Parallel Programs”. In: *Proceedings of the First Israeli Conference on Computer Systems Engineering*. May 1986.

[141] Robert O’Callahan and Jong-Deok Choi. “Hybrid Dynamic Data Race Detection”. In: *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’03. San Diego, California, USA: ACM, 2003, pp. 167–178.

[142] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. “Kendo: Efficient Deterministic Multithreading in Software”. In: *ASPLOS*. Washington, DC, USA, 2009, pp. 97–108. DOI: http://doi.acm.org/10.1145/1508244.1508256.

[143] *OpenMP application program interface, version 3.0*. May 2008.

[144] Y. Oyama, K. Taura, and A. Yonezawa. "Executing Parallel Programs With Synchronization Bottlenecks Efficiently". In: *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*. Sendai, Japan, 1999, pp. 182–204.

[145] Richard C. Paige and Clyde P. Kruskal. "Parallel Algorithms for Shortest Path Problems". In: *Int. Conference on Parallel Processing*. 1985, pp. 14–20.

[146] Soyeon Park et al. "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors". In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, 2009, pp. 177–192. DOI: http://doi.acm.org/10.1145/1629575.1629593.

[147] Harish Patil et al. "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. Toronto, Ontario, Canada: ACM, 2010, pp. 2–11. ISBN: 978-1-60558-635-9. DOI: 10.1145/1772954.1772958. URL: http://doi.acm.org/10.1145/1772954.1772958.

[148] Wolfgang J. Paul, Uzi Vishkin, and Hubert Wagener. "Parallel Dictionaries in 2-3 Trees". In: *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*. Barcelona, Spain, 1983, pp. 597–609.

[149] Gilles Pokam et al. "Architecting a Chunk-based Memory Race Recorder in Modern CMPs". In: New York, New York, 2009, pp. 576–585. DOI: http://doi.acm.org/10.1145/1669112.1669183.

[150] Nagavamsi Ponnekanti and Hanuma Kodavalla. "Online index rebuild". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. 2000, pp. 529–538.

[151] Kevin Poulsen. *Tracking the blackout bug*. 2004. URL: http://www.securityfocus.com/news/8412 (visited on 11/02/2016).

[152] Eli Pozniansky and Assaf Schuster. "MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles". In: *Concurr. Comput. : Pract. Exper.* 19.3 (Mar. 2007), pp. 327–340. ISSN: 1532-0626. DOI: 10.1002/cpe.v19:3. URL: http://dx.doi.org/10.1002/cpe.v19:3.

[153] Christoph von Praun and Thomas R. Gross. "Object Race Detection". In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '01. Tampa Bay, FL, USA: ACM, 2001, pp. 70–82.

[154] Raghavan Raman et al. "Efficient Data Race Detection for Async-finish Parallelism". In: *Proceedings of the First International Conference on Runtime Verification*. RV'10. St. Julians, Malta: Springer-Verlag, 2010, pp. 368–383. ISBN: 3-642-16611-3, 978-3-642-16611-2. URL: http://dl.acm.org/citation.cfm?id=1939399.1939430.

[155] Raghavan Raman et al. “Scalable and Precise Dynamic Datarace Detection for Structured Parallelism”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, 2012, pp. 531–542. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254127. URL: http://doi.acm.org/10.1145/2254064.2254127.

[156] Martin C. Rinard and Monica S. Lam. “The Design, Implementation, and Evaluation of Jade”. In: 20 (3 1998), pp. 483–545. ISSN: 0164-0925. DOI: http://doi.acm.org/10.1145/291889.291893.

[157] Michiel Ronsse and Koen De Bosschere. “RecPlay: A Fully Integrated Practical Record/Replay System”. In: 17 (2 1999), pp. 133–152. ISSN: 0734-2071. DOI: http://doi.acm.org/10.1145/312203.312214.

[158] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. “A Simple Load Balancing Scheme for Task Allocation in Parallel Machines”. In: *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’91. Hilton Head, South Carolina, USA: ACM, 1991, pp. 237–245. ISBN: 0-89791-438-4. DOI: 10.1145/113379.113401. URL: http://doi.acm.org/10.1145/113379.113401.

[159] Karl Rupp. *40 Years of Microprocessor Trend Data*. 2015. URL: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data (visited on 06/21/2017).

[160] Yehoshua Sagiv. “Concurrent Operations on B-trees with Overtaking”. In: *Journal of Computer and System Sciences* 33.2 (1986), pp. 275–296.

[161] Yehoshua Sagiv. “Concurrent Operations on B*-trees with overtaking.” In: *Proc. of the 4th Annual Symp. on the Principles of Database Systems*. 1999, pp. 28–37.

[162] B. Samadi. “B-Trees in a System with Multiple Users”. In: *Inform. Proc. Letters* 5.4 (1976), pp. 107–112.

[163] Peter Sanders. “Randomized Priority Queues for Fast Parallel Access”. In: *Journal of Parallel Distributed Computing* 49.1 (1998), pp. 86–97.

[164] Stefan Savage et al. “Eraser: A Dynamic Data Race Detector for Multi-threaded Programs”. In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP ’97. Saint Malo, France: ACM, 1997, pp. 27–37. ISBN: 0-89791-916-5. DOI: 10.1145/268998.266641. URL: http://doi.acm.org/10.1145/268998.266641.

[165] Tao B. Schardl et al. “The Cilkprof Scalability Profiler”. In: *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*. SPAA ’15. Portland, Oregon, USA: ACM, June 2015, pp. 89–100. ISBN: 978-1-4503-3588-1. URL: http://doi.acm.org/10.1145/2755573.2755603.

[166] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: Data Race Detection in Practice". In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA '09. New York, New York, USA: ACM, 2009, pp. 62–71. ISBN: 978-1-60558-793-6. DOI: 10.1145/1791194.1791203. URL: http://doi.acm.org/10.1145/1791194.1791203.

[167] Nir Shavit and Asaph Zemach. "Combining Funnels: a dynamic approach to software combining". In: *J. Parallel Distrib. Comput.* 60.11 (Nov. 2000), pp. 1355–1387. ISSN: 0743-7315. DOI: 10.1006/jpdc.2000.1621. URL: http://dx.doi.org/10.1006/jpdc.2000.1621.

[168] Nir Shavit and Asaph Zemach. "Diffracting trees". In: *ACM Trans. Comput. Syst.* 14.4 (Nov. 1996), pp. 385–428. ISSN: 0734-2071. DOI: 10.1145/235543.235546. URL: http://doi.acm.org/10.1145/235543.235546.

[169] Julian Shun et al. "Brief announcement: the Problem Based Benchmark Suite". In: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '12. 2012.

[170] Julian Shun et al. "Reducing Contention Through Priority Updates". In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: ACM, 2013, pp. 299–300. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442554. URL: http://doi.acm.org/10.1145/2442516.2442554.

[171] V. Srinivasan and Michael J. Carey. "Performance of B-tree concurrency control algorithms". In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. 1991, pp. 416–425.

[172] Guy L. Steele Jr. "Making Asynchronous Parallelism Safe for the World". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA: ACM, 1990, pp. 218–231. ISBN: 0-89791-343-4. DOI: 10.1145/96709.96731. URL: http://doi.acm.org/10.1145/96709.96731.

[173] Rishi Surendran and Vivek Sarkar. "Automatic Parallelization of Pure Method Calls via Conditional Future Synthesis". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: ACM, 2016, pp. 20–38. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984035. URL: http://doi.acm.org/10.1145/2983990.2984035.

[174] Rishi Surendran and Vivek Sarkar. "Brief Announcement: Dynamic Determinacy Race Detection for Task Parallelism with Futures". In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '16. Pacific Grove, California, USA: ACM, 2016, pp. 95–97. ISBN: 978-1-4503-4210-0. DOI: 10.1145/2935764.2935815. URL: http://doi.acm.org/10.1145/2935764.2935815.

[175] Herb Sutter. “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”. In: *Dr. Dobb's Journal* 30.3 (2005).

[176] Nathan R. Tallent and John M. Mellor-Crummey. “Effective Performance Measurement and Analysis of Multithreaded Applications”. In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '09. Raleigh, NC, USA: ACM, 2009, pp. 229–240. ISBN: 978-1-60558-397-6. DOI: 10.1145/1504176.1504210. URL: http://doi.acm.org/10.1145/1504176.1504210.

[177] Nathan R. Tallent and John M. Mellor-Crummey. “Identifying Performance Bottlenecks in Work-Stealing Computations”. In: *Computer* 42.12 (2009), pp. 44–50. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2009.396.

[178] Robert Endre Tarjan. “Applications of Path Compression on Balanced Trees”. In: *Journal of the ACM* 26.4 (Oct. 1979), pp. 690–715. ISSN: 0004-5411. DOI: 10.1145/322154.322161. URL: http://doi.acm.org/10.1145/322154.322161.

[179] Robert Endre Tarjan. “Efficiency of a Good But Not Linear Set Union Algorithm”. In: *Journal of the ACM* 22.2 (Apr. 1975), pp. 215–225.

[180] Sağnak Taşırlar and Vivek Sarkar. “Data-Driven Tasks and Their Implementation”. In: *Proceedings of the 2011 International Conference on Parallel Processing*. ICPP '11. Taipei City, Taiwan: IEEE Computer Society, 2011, pp. 652–661.

[181] *Intel(R) Threading Building Blocks*. Available from http://www.threadingbuildingblocks.org/documentation.php. Intel Corporation. 2009.

[182] Intel Cilk Plus Development Team. *Intrinsics for Low Overhead Tool Annotations*. Tech. rep. Nov. 2011.

[183] Athanasios K. Tsakalidis. “Maintaining Order in a Generalized Linked List”. In: *Acta Inf.* 21.1 (June 1984), pp. 101–112. ISSN: 0001-5903. DOI: 10.1007/BF00289142. URL: http://dx.doi.org/10.1007/BF00289142.

[184] Robert Utterback et al. “Processor-Oblivious Record and Replay”. In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '17. Austin, Texas, USA: ACM, 2017.

[185] Robert Utterback et al. “Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '16. Pacific Grove, California, USA: ACM, 2016, pp. 83–94. ISBN: 978-1-4503-4210-0. DOI: 10.1145/2935764.2935801. URL: http://doi.acm.org/10.1145/2935764.2935801.

[186] Kaushik Veeraraghavan et al. “DoublePlay: Parallelizing Sequential Logging and Replay”. In: *ASPLOS*. Newport Beach, California, USA, 2011, pp. 15–26. DOI: http://doi.acm.org/10.1145/1950365.1950370.

[187] Haixun Wang et al. “Dual Labeling: Answering Graph Reachability Queries in Constant Time”. In: *22nd International Conference on Data Engineering (ICDE'06)*. Apr. 2006, pp. 75–75. DOI: 10.1109/ICDE.2006.53.

[188] K. B. Wheeler, R. C. Murphy, and D. Thain. “Qthreads: An API for programming with millions of lightweight threads”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. Apr. 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536359.

[189] Min Xu, Rastislav Bodik, and Mark D. Hill. “A “Flight Data Recorder” for Enabling Full-system Multiprocessor Deterministic Replay”. In: San Diego, California, 2003, pp. 122–135. DOI: http://doi.acm.org/10.1145/859618.859633.

[190] Zhemin Yang et al. “ORDER: Object Centric Deterministic Replay for Java”. In: *Proceedings of the USENIX Annual Technical Symposion*. Portland, OR, 2011, pp. 30–30.

[191] Yuan Yu, Tom Rodeheffer, and Wei Chen. “RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking”. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 221–234.

[192] Christopher S. Zakian et al. “Concurrent Cilk: Lazy Promotion from Tasks to Threads in C/C++”. In: *Languages and Compilers for Parallel Computing: 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers*. Ed. by Xipeng Shen, Frank Mueller, and James Tuck. Cham: Springer International Publishing, 2016, pp. 73–90. ISBN: 978-3-319-29778-1. URL: http://dx.doi.org/10.1007/978-3-319-29778-1_5.