PERFORMANCE ENGINEERING OF MULTICORE SOFTWARE: DEVELOPING A SCIENCE OF FAST CODE FOR THE POST-MOORE ERA

by

TAO BENJAMIN SCHARDL

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

Author . . . .
Department of Electrical Engineering and Computer Science
August 31, 2016

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

Accepted by . . . .
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Performance Engineering of Multicore Software:
Developing a Science of Fast Code for the Post-Moore Era

by

Tao Benjamin Schardl

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

The end of Moore's Law, which experts predict to occur in as few as 5 years, means that even average programmers will need to be able to write fast code. Software performance engineering offers great promise to provide computer performance gains in the post-Moore era, but developing efficient software today requires substantial expertise and arcane knowledge of hardware and software systems. Multicore processors are particularly challenging to use efficiently, because doing so requires programmers to engage in parallel programming and to deal with nondeterministic program behavior and parallel scalability concerns.

I contend that we can remedy the *ad hoc* and unprincipled nature of software performance engineering by creating simple and integrated programming technologies for writing fast code. This thesis studies how such technologies can be built by examining nine artifacts that enable principled approaches to tackling nondeterminism and scalability concerns in writing efficient multicore software. Five artifacts develop programming models and theories of performance for writing multicore programs that are efficient both in theory and in practice:

- PBFS, a work-efficient parallel breadth-first search algorithm.
- The Prism chromatic-scheduling algorithm, which executes dynamic data-graph computations deterministically in parallel.
- Ordering heuristics for parallel greedy graph coloring algorithms.
- The pedigree mechanism and DotMix algorithm for generating pseudorandom numbers deterministically in parallel in dynamic multithreaded programs.
- The Cilk-P concurrency platform, which provides linguistic and runtime support for deterministic on-the-fly pipeline parallelism.

Three artifacts strive to embed abstract programming and performance models into tools and compilers:

- Cilkprof, a profiler that efficiently measures how each call site in a Cilk program contributes to the program's scalability.
- Rader, a provably good race detector for Cilk programs that use reducer hyperobjects.
- The Tapir compiler intermediate representation, which enables existing compiler optimizations for serial code to optimize across parallel control flow with minimal changes.

The final artifact tackles the complexity of creating efficient diagnostic tools:

- CSI, a framework that provides comprehensive static instrumentation for efficient dynamic-analysis tools.

Together, these artifacts contribute to developing a more coherent science of fast code for multicores than exists today.

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

Acknowledgments

There are numerous people who deserve tremendous thanks for shaping my development as a researcher and as a person, without whom this thesis would not have been possible. I will do my best to thank as many as of them here as I can.

First and foremost, thanks to my advisor, Charles E. Leiserson, for being a phenomenal mentor, teacher, and collaborator. Although I had had some experience doing research before I met Charles, I never knew just how invigorating and fun research can really be until we started working together. His shining example inspires me each day to tackle every problem head-on, to follow every line of inquiry as far as I can, and to strive to be a better writer and teacher.

I would like to thank Saman Amarasinghe and Guy E. Blelloch, for their invaluable feedback and support over the years, as well as their service on my thesis committee. Saman and I have talked many times about the practical side of software performance engineering, including the variety of strange performance anomalies that arise when writing fast code and strategies and techniques for teaching students how to write efficient software. Guy and I have had many discussions on the theoretical side of software performance engineering, and thanks to Guy and his research group, I have had the pleasure of seeing many beautiful parallel algorithms and analysis techniques.

Erik D. Demaine deserves thanks for his advice during my undergraduate career at MIT and for entertaining many research discussions in programming languages, computational geometry, and data structures during my undergraduate career and my early graduate career. Charles, Saman, Guy, and Erik also all deserve many thanks for the indispensable career advice they given me.

Thanks to all of my coauthors who helped me produce the work presented in this thesis.

- Jim Sukha coauthored the DPRNG and Cilk-P papers. I also thank Jim for the many research discussions we have had on pedigrees, random number generators, reducers, pipelining, and Cilk technology.
- I-Ting Angelina Lee coauthored the Cilk-P, Cilkprof, Rader, and CSI papers. Angelina also deserves many thanks for showing me the internals of the Cilk runtime and for research discussions on many topics, including pipelining, reducers, race detection, runtime systems, and compiler instrumentation.
- Zhunping (Justin) Zhang coauthored the Cilk-P paper. I also thank Justin for great discussions on performance-engineering the PARSEC benchmarks using Cilk technology.
- William Hasenplaugh coauthored the Prism and graph-coloring papers. Over my graduate career, Will and I have had many conversations on provably efficient algorithms and proof techniques I had not seen before, for which I am thankful.
- Tim Kaler coauthored the Prism and graph-coloring papers as well. Tim also deserves many thanks for several great conversations on various topics, including efficient data structures, reducers, pedigrees, performance analysis, how to teach software performance engineering, and how to write fast code in the cloud.
- Bradley C. Kuszmaul coauthored the Cilkprof paper and the Life after Moore's Law article. In addition, many thanks to Bradley for collaborating on the textbook and the matrix-multiplication case study, as well as for showing me many esoteric features of programming languages and many brilliant software-performance-engineering tricks.
- William M. Leiserson coauthored the Cilkprof paper. I also thank Will for teaching me about Cilk technology early in my graduate career, for his work in developing the

prototype Cilkprof Pin tool, and for ongoing conversations about parallel programming and programming languages.

- William (Billy) Moses coauthored the Tapir paper, and I commend Billy for his phenomenal efforts in developing the prototype implementation of Tapir in LLVM. Billy also deserves thanks for many discussions on reducers and programming languages.
- Tyler Denniston and Damon Doucet coauthored the CSI paper. I thank both Tyler and Damon for their insights on the design and implementation of CSI. Tyler also deserves thanks for his ongoing efforts in upstreaming CSI into mainstream LLVM.
- Joel S. Emer, Butler W. Lampson, Daniel Sanchez, and Neil C. Thompson all coauthored the Life after Moore's Law article. Thanks to Joel and Daniel for teaching me about modern computer architecture and the impact of Moore's Law. Butler deserves thanks for teaching me about large components and the development of large systems. Many thanks to Neil for teaching me about the economics related to Moore's Law.

In addition to the coauthors and collaborators mentioned so far, I also thank everyone I have collaborated with on different projects and research meetings over the years. Thanks to all of the past and present members of the Supertech research group, especially Kunal Agrawal, Aydın Buluç, Maryam Mehri Dehnavi, Jeremy T. Fineman, Matteo Frigo, Miriam Gershenson, Predrag Gruevski, Shahin Kamali, Matthew Kilgore, Edya Ladan-Moses, Nasro Min-Allah, Eka Palamadai Natarajan, Warut Suksompong, Yuan Tang, James J. Thomas, and Kerry Xing. Thanks to Intel, especially Pablo Halpern, Barry Tannenbaum, and other members of the Cilk team, for discussions on Cilk technology and for incorporating some of our ideas into Intel Cilk Plus. Many thanks to Ronald L. Rivest for his service on my RQE committee and his feedback on the DPRNG paper. Thanks to the participants of various open problem sessions I have attended, especially Zachary Abel, Martin L. Demaine, Sarah Eisenstat, Jayson Lynch, and Isaac Shapiro-Ellowitz. I thank Guy L. Steele Jr. for interesting discussions on fast parallel random-number generation. I thank my collaborators at CMU, including Phillip B. Gibbons, Julian Shun, Harsha Vardhan Simhadri, Kanat Tangwongsan, and the participants of the Parlay research meetings, for showing me many beautiful algorithms. I thank the participants of the parallel video-coding project, especially Daewook Kim, Vivienne Sze, and various visitors from Foxconn. Thanks to Robert C. Miller and Genghis Chau for discussions on visualizing the output of different Cilk tools. I thank our Google sponsors, especially Kostya Serebryany, for their feedback and guidance on the CSI project. Arvind, Christian J. Bell, Adam Chlipala, and Mohsen Lesani deserve thanks for their feedback on Tapir and formal semantics. Thanks to everyone who contributed to discussions on Life after Moore's Law, especially Arvind, Jack Dennis, and Srini Devadas.

Finally, I thank all of my friends and family, especially my sister, Anisa, my brother-in-law, Matt,[1] and my parents, Chris and Kuey-Chu. Without their inexorable love, support, and encouragement, I never would have made it through my undergraduate and graduate student careers at MIT.

[1]Perhaps more commonly known as "Snowflake."

Contents

Chapter 1

Introduction

With a growing number of indications that the end of Moore's Law is imminent — meaning that advances in silicon fabrication technology can no longer sustain the historical exponential growth of computer performance — even average programmers will soon need to be able to write efficient software. Although programmers can find considerable gains in computer performance through "software performance engineering," writing efficient software today requires substantial expertise and arcane knowledge. I contend that the process of writing efficient software can be made accessible to average programmers by building simple and integrated software-performance-engineering technologies. In particular, building these technologies serves to develop a more coherent science of fast code than exists today. This thesis examines the development of such technologies by examining nine artifacts that support principled approaches to tackling complex issues in writing fast multicore programs.

Software performance — how quickly code runs and how efficiently it uses computing resources, such as memory, energy, network bandwidth, secondary storage, etc. — dictates the cost of running an application on a computing device. Every software application requires some time and computing resources in order to run, and new software functionality requires additional computing resources to run alongside existing applications. Optimizing a program for performance ensures that computing devices can meet the resource requirements of that program. Recently, cloud computing [14, 15, 23, 138, 166] has generated monetary incentives for optimizing software performance. The monetary cost of running an application in the cloud scales with its running time [16]. Minimizing the running time of the application therefore minimizes the cost of running that application in the cloud.

Despite the monetary and human costs of slow code, ***software performance engineering*** — the rewriting code to run faster and use computing resources more efficiently — is a notoriously difficult task [221]. Attempting to improve a program's performance tends to complicate its codebase, making the code difficult to understand, maintain, and debug. Furthermore, programmers often demonstrate a poor intuitive understanding of where programs spend their execution time. As a result, programmers can waste significant time and energy making code more complicated without actually improving its performance. These issues with software performance engineering have been known since the 1970's and 1980's, when several famous criticisms were made of programmers attempting to improve software performance:

> Premature optimization is the root of all evil.
>
> —*Donald Knuth, 1979 [221]*

Version	*Implementation*	*Running time (s)*	*GFLOPS*	*Absolute speedup*	*Relative speedup*	*Fraction of peak*
1	Python	25,552.48	0.005	1	—	0.00 %
2	Java	2,372.68	0.058	11	10.8	0.01 %
3	C	542.67	0.253	47	4.4	0.03 %
4	Parallel loops	69.80	1.969	366	7.8	0.24 %
5	Parallel divide-and-conquer	3.80	36.180	6,727	18.4	4.33 %
6	+ vectorization	1.10	124.914	23,224	3.5	14.96 %
7	+ AVX intrinsics	0.41	337.812	62,806	2.7	40.45 %
8	Strassen	0.38	361.177	67,150	1.1	43.24 %

Figure 1-1: Performance gains from iteratively performance-engineering a program on a modern multicore machine to multiply two 4096×4096 matrices of double-precision floating-point numbers. "*Time*" is the running time in seconds of the particular code. "*GFLOPS*" is the giga-FLOPS — billions of floating-point operations per second — that the code achieved. "*Absolute speedup*" is the ratio of the running time of the code to that of the Python code in Version 1 of the table. "*Relative speedup*" is the ratio of the running time of the code to the running time of the code on the preceding line of the table. "*Fraction of peak*" is the percentage of the computer's maximum double-precision floating-point capability, which is 835 GFLOPS. The "*GFLOPS*" and "*Percent of peak*" values for Strassen are computed based on time, as if Strassen actually performed $2 \cdot 4096^3$ floating-point operations in its running time. Each running time is the minimum of 5 runs on an Amazon AWS `c4.8xlarge` spot instance, which is a dual-socket Intel Xeon E5-2666 v3 system with a total of 60 GiB of memory. Each Xeon is a 2.9 GHz 18-core CPU with a shared 25 MiB L3-cache. Each processor core has a 32 KiB private L1-data-cache and a 256 KiB private L2-cache. The machine was running Fedora 22, using version 4.0.4 of the Linux kernel. The Python version was executed using Python 2.7.9. The Java version was compiled and run using OpenJDK version 1.8.0_51. All other versions were compiled using GCC 5.2.1 20150826.

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity.
—*William Wulf, 1979 [410]*

The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization — For experts only: Don't do it yet.
—*Michael A. Jackson (quoted by Jon Bentley, 1988 [39])*

These criticism were originally uttered in response to programmers using `goto` statements in complicated ways in efforts to improve software efficiency. These efforts would produce codes that are difficult to reason about and debug. Although programmers today typically use structured control flow, such as conditionals and loops, instead of `goto` statements, the act of writing fast code still tends to produce complicated code that is difficult to understand, maintain, and debug.

To illustrate the complexity and payoff of software performance engineering today, consider a simple example: matrix multiplication. Figure 1-1 presents the results of a case study my colleagues[1] and I undertook to performance-engineer a program that multiplies two 4096×4096 matrices on a shared-memory multicore machine.[2] Version 1 reflects what a programmer today might initially implement to solve the problem: multiply the two matrices using three nested loops in Python, one of today's most popular high-level programming languages. As the table shows, this code takes 7 hours to run and achieves less than 1/150,000 of the machine's peak performance.

[1]I collaborated with Bradley C. Kuszmaul and Charles E. Leiserson on this case study.
[2]This case study was originally inspired by Saman Amarasinghe.

Starting from the Python implementation, we engineered this software for performance using a variety of techniques. Simply choosing a more efficient programming language speeds up this calculation dramatically. For example, coding it in Java (Version 2) produces a speedup of 10.8 ×, and coding it in C (Version 3) produces an additional speedup of 4.4 ×, or 47 times faster than the original Python code. "Form-fitting" the matrix code to take advantage of the hardware makes it still faster. By parallelizing the code to run on all 16 of the processor cores (Version 4), exploiting the cache hierarchy (Version 5), vectorizing the code (Version 6), and exploiting Intel's special AVX chip instructions (Version 7), the matrix-multiplication task can be solved in 0.41 seconds. By using Strassen's celebrated algorithm [375] for matrix multiplication, the highly optimized matrix-multiplication code in Version 7 of Figure 1-1 can be improved about 10 % to achieve a running time of 0.38 seconds, a 67,000 × speedup over the original Python code.

This case study also illustrates the difficulty of writing fast code. Today, there are three general techniques to improve software efficiency, each of which complicates programming. (1) Adopt low-level programming languages and avoid high-level programming technologies that trade off performance for ease of programming. (2) Refactor code to exploit architectural features, including parallelism, cache hierarchy, and vectorization. (3) Rewrite code to use more efficient algorithms. When all of these techniques are applied in the case study, the resulting efficient parallel Strassen implementation (Version 8) is over 850 lines of code, which is 40 times the number of lines in the original Python implementation (Version 1). It took us over 100 hours of programmer effort to produce the Strassen code, even though we availed ourselves of the latest software-performance-engineering technology produced by recent research. Of course, for the specific problem of matrix multiplication, one can simply call an already optimized routine, such as DGEMM from the Intel Math Kernel Library (MKL) [200], which is almost as fast as Version 8. For arbitrary pieces of code where such libraries are not available, however, programmers must use these techniques themselves and deal with the concomitant programming difficulties. As the study shows, there exists a vast distance in both program efficiency and programming difficulty between simple code and fast code.

Software performance engineering and Moore's Law

Should average programmers worry about writing fast code? Programmers today often don't concern themselves with engineering software for performance, in part because of "Moore's Law." In 1965 Intel founder Gordon Moore observed the steady trend of transistor miniaturization [287], and in 1975 he predicted that the number of transistors per computer chip would double every two years [288]. This trend was christened ***Moore's Law*** by Caltech professor Carver Mead (although the precise origins are actually a bit murky [102]) and has held since 1975. Until about 2006, smaller transistors were also faster and more energy efficient [109], which enhanced the speed and capacity of computers. Moore's Law is responsible for cellphones today that are more powerful than the room-sized computers from just 25 years ago, for providing nearly half the world's people with access to all the world's information on the Internet [202], and for drug discoveries enabled by powerful supercomputers. With Moore's Law providing programmers with "free" computer performance, programmers found little economic incentive to struggle with the complex task of writing fast code and instead focused on minimizing the development time of their applications.

Because of both fundamental physical limits and because of the economics of chip manufacturing, however, it appears that Moore's Law will end soon. Technology leaders, such as Robert Colwell [95] and the *International Technology Roadmap for Semiconductors* [203] have predicted the demise of Moore's Law to occur as early as 2020. Even Intel, the leader in semiconductor-device technology, has acknowledged a significant attenuation in the rate of shrinking device sizes [201, p. 14].

After the demise of Moore's Law, programmers must find other opportunities for computer performance to grow. The case study in Figure 1-1 suggests that programmers can find substantial performance gains by performance-engineering software. For example, consider a simple conception of Moore's Law that says that computers get twice as powerful every two years.[3] Then the speedup factor of 67,000 achieved by the Strassen implementation (Version 8) over the original Python program (Version 1) amounts to over 32 years of Moore's Law improvements. Other promising results can be found in studies in performance-engineering large graph-processing codes [320] and in performance-engineering the "connectomics" pipeline for processing images of brain tissue to determine neural connectivity [253, 353]. Although these problems are often thought to require large clusters of machines to solve, these studies successfully performance-engineer these codes to run more efficiently on single multicore machines. But performance-engineering software for today's machines still requires substantial expertise and arcane knowledge, as the case study in Figure 1-1 illustrates. How can we expect average programmers to do such optimizations?

Previous research on software performance engineering

Decades of research effort have gone into mitigating the complexity of software performance engineering. This research effort has produced an array of technologies that support principled approaches to writing and reasoning about fast code, including programming languages and frameworks, compilers, performance analysis tools, theoretical models of performance, provably efficient algorithms, and methodologies for developing efficient software.

Several fundamental tools and methodologies for writing efficient software find their origins in the 1970's and 1980's. In 1971, Knuth developed the first program ***profiler*** to measure how frequently a program executed each of its statements [220]. Knuth and other programmers used the resulting program profiles to identify bottlenecks in the execution time of a program and to direct their optimization efforts towards those bottlenecks. In 1975, Wulf *et al.* developed one of the earliest optimizing compilers [411], which empowered programmers to use higher-level programming-language constructs without sacrificing undue performance. In 1982, Bentley authored a short treatise on software performance engineering, entitled *Writing Efficient Programs* [38], which presents a set of simple rules for programmers to apply to their source code to optimize their program's performance. Although some of these rules have been encoded into optimizing compilers or have grown obsolete with advances in computer hardware, many of these rules are still applicable today.

Thanks to ongoing research and development efforts, programmers today can avail themselves of an abundance of diagnostic tools, compilers, and programming models to simplify the task of performance-engineering software for modern machines. A host of dynamic-analysis tools, such as race detectors [123, 124, 134, 280, 281, 343, 351], memory checkers [31, 177, 350], cache simulators [121, 377, 407], call-graph generators [171, 205], code-coverage analyzers [390, 397], and performance profilers [171, 329, 401], have been built that allow programmers to study how an executing program spends its time and computing

[3]In fact, computer performance has grown at slower and faster rates over time [181], but this thesis uses this commonly accepted definition for simplicity.

resources. Modern multicore processors contain hardware performance counters that diagnostic tools can employ to gain detailed insight into how a program utilizes the hardware. Mainstream compilers, including GCC [369], ICC [198], and LLVM [232], are all optimizing compilers that are designed to handle multiple front-end programming languages and multiple back-end machine architectures. "Just-in-time" compilers have been created that compile and optimize a program as it executes (see [25] for a history). These compiler technologies allow a wide variety of software to enjoy the benefits of compilation and automatic optimization. Tools such as autotuners [18, 19, 91, 142, 322, 400, 404] and polyhedral optimization frameworks [63, 172] have been built to perform elaborate optimizations on specific types of programs or program structures. Parallel building blocks [357] and "dynamic multithreading concurrency platforms" (described in Section 1.1) have been built that temper the complexity parallel programming. Domain-specific languages have emerged that simplify the task of writing efficient programs in specific problem domains. The Halide programming language [323] aims to enable performance engineers to quickly iterate through a variety of image-processing-pipeline implementations to zero in on the fastest one. The Julia programming language [44] simplifies the task of writing fast programs in technical computing. Several frameworks have been created to support efficient computations on graphs, including GraphLab [264, 265] Pregel [269], Galois [298, 299], PowerGraph [165], Ligra [358, 361], and GraphChi [228].

Immense research effort has also gone into creating efficient algorithms and data structures for numerous problems, as well as creating analytical techniques and performance models for justifying an algorithm's efficiency [9, 101, 223, 349]. Although most algorithms and data structures are designed to optimize the serial running time of a computation, theoretical performance models have been developed to address particular features of modern multicore hardware. We now have many good performance models for parallel machines, including the parallel random-access machine (PRAM) model and its variants (see [214] for a survey), the bulk-synchronous model [393], the work-span model [100, Ch. 27], the logP model [105], fixed-connection networks [244, Sec. 1.1], and even Boolean circuits [396]. Meanwhile, performance models, such as external memory model and its variants [5, 145] and the parallel external memory model [21] allow programmers to analyze performance in terms of memory accesses, which are relatively expensive operations on modern machines. Theoretical performance models have also emerged to account for high-latency operations such as file I/O in parallel programs [291].

Given these numerous developments, why does software performance engineering continue to be an art that requires considerable expertise and arcane knowledge to carry out? From my own research experience, I find that many existing performance-engineering technologies are difficult to use because they are not *integrated*. For example, many theoretically efficient algorithms and data structures described in the literature have never been implemented, leaving the question of whether their theoretical efficiency can be borne out in practice unresolved. Furthermore, most algorithms are still designed to minimize only their asymptotic serial running time, and relatively little attention is paid to "form-fitting" these algorithms to efficiently utilize modern multicore hardware. Meanwhile, existing diagnostic tools and compilers are often difficult to use because their behavior does not reflect the simple programming and performance models used to write efficient software. Diagnostic tools can produce results that defy interpretation in terms of the models, while compilers can fail to perform optimizations that are simple and straightforward in light of the programming model. Although some developments, such as the dynamic multithreading concurrency platforms described in Section 1.1, combine sets of performance-engineering

Artifact	*Simple programming models*	*Theories of performance*	*Diagnostic tools*	*Compilers*
PBFS	●	●		
Prism	●	●		
Color	●	●		
DPRNG	●			
Cilk-P	●	●		
Cilkprof	●	●	●	●
Rader	●		●	●
Tapir	●			●
CSI	●		●	●

Figure 1-2: Illustration of the high-level software-performance-engineering technologies used or developed by each artifact presented in this thesis. The first column lists the artifacts in this thesis, while the remaining columns identify high-level software-performance-engineering technologies: column two denotes simple programming models, column three denotes theories of performance, column four denotes diagnostic tools, and column five denotes compilers. Each row corresponds to a thesis artifact. A circle indicates that a particular thesis artifact used or developed a particular technology. "PBFS" identifies my work on a work-efficient parallel breadth-first search algorithm (Chapter 3). "Prism" identifies my work on the Prism chromatic-scheduling algorithm for dynamic data-graph computations (Chapter 4). "Color" identifies my work on ordering-heuristics for parallel graph coloring (Chapter 5). "DPRNG" identifies my work on deterministic parallel random-number generation (Chapter 6). "Cilk-P" identifies my work on the Cilk-P concurrency platform for on-the-fly pipeline parallelism (Chapter 7). "Cilkprof" identifies my work on the Cilkprof scalability profiler (Chapter 8). "Rader" identifies my work on the Rader race detector for Cilk programs that use reducer hyperobjects (Chapter 9). "Tapir" identifies my work on the Tapir compiler intermediate representation for dynamic multithreaded programs (Chapter 10). "CSI" identifies my work on the CSI compiler instrumentation framework (Chapter 11).

technologies, further integration of these technologies is needed to reduce the expertise and arcane knowledge required to write fast code.

Contributions

I contend that we can remedy the *ad hoc* and unprincipled nature of software performance engineering by transforming the writing of fast code into a more coherent engineering science. To do so, the disparate technologies that support software performance engineering must be integrated and new technologies must be developed. Programming models, theories of performance, compilers, and tools must all work together to reduce the complexity of writing fast code and reasoning about its behavior and efficiency. As long as programmers must reconcile the diverse assumptions and behaviors of different performance-engineering technologies themselves, writing fast code will continue to be more of an art than a science. Only by integrating the various aspects of software development can performance engineers hope to realize the central promise of science: the ability to make testable predictions about how things behave. Today, programmers generally must choose between writing simple code they can understand or writing fast code. I contend that we can reduce the distance between simple code and fast code by building integrated software-performance-engineering technologies and developing a more coherent science of efficient software.

This thesis explores how simple and integrated software-performance-engineering technologies can be built by studying nine artifacts that tackle complex issues facing performance engineers when writing efficient multicore software. Figure 1-2 illustrates the set of high-level software-performance-engineering technologies that are used or developed by each thesis artifact. As the figure shows, the first five artifacts develop simple programming models and theories of performance that enable principled approaches to reasoning about the behavior and efficiency of multicore software. The remaining four artifacts focus on the diagnostic tools and compilers in the software-development environment. Three of these artifacts work to embed the abstract programming models and theories of performance for efficient multicore software into these diagnostic tools and compilers. This embedding strives to ensure that the behavior of the development environment conforms to the programmer's reasoning about the program's performance and behavior. The final artifact introduces a simple programming model for creating new efficient diagnostic tools. Together, these artifacts examine how software-performance-engineering technologies can be built and integrated to support principled, scientific approaches to reasoning about the behavior and performance of fast code for multicores.

The remainder of this chapter overviews the challenges in writing fast multicore software and how the nine artifacts presented in this thesis develop technologies to support principled, scientific approaches to addressing these challenges. Section 1.1 describes "shared-memory multicore machines" and the major issues of "nondeterminism" and "scalability" that complicate multicore-software performance engineering. This section also describes prior work on "dynamic multithreading" to address some of these issues. Section 1.2 describes the first five artifacts and how they develop simple programming models that admit theories of performance for writing and reasoning about efficient multicore software. Section 1.3 describes the last four artifacts, which look to enhance the software-development environment to support principled approaches to reasoning about the behavior and performance of multicore programs. Section 1.4 reviews the statement of this thesis, and Section 1.5 outlines the remaining chapters of this thesis.

1.1 Shared-memory-multicore programming

Most of the artifacts presented in this thesis tackle the complexity of writing efficient software for a ***shared-memory multicore machine*** (or simply a ***multicore***), which is a computer that contains multiple general-purpose processor cores that share a common main memory. This section describes why studying shared-memory multicores will be important in the post-Moore era. This section also describes parallel programming, one of the largest and most difficult problems in programming multicores, and the main concerns of "nondeterminism" and "scalability" that make writing fast multicore programs particularly challenging. I briefly outline "dynamic multithreading" technology,[4] which makes substantial progress in addressing nondeterminism and scalability issues in multicore programming. Chapter 2 describes dynamic multithreading in greater detail.

One reason for why multicore programming will be important in the post-Moore era comes from their ubiquity in the computing environment. Multicores entered the marketplace around the mid-2000's, when processor manufacturers could no longer run transistors at higher frequencies due to physical limitations on processor hardware, specifically, on the power density of silicon technology. Since then, multicore processors have proliferated to

[4]Also called ***task parallelism***.

become the “bargain” processor component that is widely employed in building a variety of computing devices, large and small. Mobile computing systems such as tablets and smartphones are typically built using a multicore processor chip. Large, complex systems, meanwhile, such as datacenters for cloud computing, grid computers, high-performance clusters, and supercomputers, are often built from collections of shared-memory multicore machines.

Shared-memory multicore machines are also simpler to program than distributed systems. Although a cluster of multicore machines has greater total capacity than any single multicore, coordinating parallel tasks in a distributed computing environment often involves moving data between machines, which incurs high performance costs. As a result, machines in distributed environments often exhibit poor utilization — software achieves little of the total potential computer performance of the cluster. In a multicore, however, main memory is shared between processor cores. As a result, data need not be copied to coordinate parallel operations, and the cost of this coordination is thereby significantly reduced. Software thus has an easier time achieving high machine utilization on a single multicore.

Furthermore, individual multicores have grown to be quite large and powerful. A single multicore system today is able to fit a whole database system within its memory [78]. A host of other applications across domains such as computational science, real-world modeling, machine learning, and image and video processing, are designed to run on multicore computers, because multicores represent a “sweet spot” in cost for performance. The aforementioned studies in large graph-processing problems and the “connectomics” image-processing pipeline demonstrate the capacity of multicores to solve problems that are typically thought to be too large for a single machine. Shun argues persuasively that single shared-memory machines are sufficient for solving many problems in large-scale computing [357].

Software often fails to realize the computer performance available within a single multicore, however. John Hennessy[5] acknowledged this trend when he said,

> We switched to multicore, but we have not made it as useful as if we had just made single-threaded processors faster.
>
> —*John Hennessy [312]*

The matrix-multiplication case study presented in Figure 1-1 reflects this trend as well. As the figure shows, software-performance-engineering effort beyond simply programming in a more efficient language is required to make the program utilize more than 1 % of the machine’s peak performance.

One reason why software tends to underutilize multicores is because multicores are complicated. A multicore contains a host of interconnected computing resources, including one or more processor chips, each of which contains multiple processor cores. Typically, a processor core is a pipelined, superscalar processing unit that includes multiple functional units, vector units, out-of-order execution, branch predictors, hardware prefetchers, hyperthreads, and one or more levels of local cache. Each processor also shares a last-level cache among all of its cores, while the machine itself contains a main memory that is shared by all of its processors. Different shared-memory multicores might contain additional computing resources, such as a network, a graphics-processing unit (GPU), a disc, or a battery. Utilizing all of these hardware features of a shared-memory multicore machine is a complex task that requires significant expertise and arcane knowledge.

[5] President of Stanford University, parallel-computing pioneer, and coauthor of *Computer Architecture: A Quantitative Approach.*

Parallel programming

Using parallel processor cores efficiently involves parallel programming, which poses one of the largest and most difficult problems in programming multicores. Parallel programming is also involved in using other multicore hardware, such as GPU's and vector units. Although ongoing important research efforts seek ways to apply these hardware systems for general computation, most software today employs these systems in an *ad hoc* and opportunistic fashion. This thesis instead focuses on programming the general-purpose processor cores, which are responsible for executing the vast majority of software running on multicores today.

To illustrate some of the challenges involved in parallel programming, let us consider writing a parallel program using Pthreads [188], a standard API for threading that is used to write many parallel programs today. The Pthreads API provides a set of library functions for creating and managing parallel operating-system threads. To write a program using Pthreads, the programmer explicitly creates each parallel thread, or Pthread, and assigns it computation to execute. The operating-system scheduler then takes care of executing these Pthreads asynchronously on processor cores. Pthreads communicate with each other and exchange data through shared memory, and programmers can coordinate updates to shared variables by employing mechanisms such as mutex locks or condition variables.

The Pthreading model forces programmers to deal with several difficult problems in reasoning about the behavior and performance of a parallel program. Pthreaded programs can behave ***nondeterministically***, based on how the operating system chooses to schedule and execute different Pthreads in any particular run of the program. Reasoning about the behavior of a Pthreaded program therefore involves considering all of the exponentially many possible timings (interleavings) of the computations running on different Pthreads, which is difficult and error-prone. Furthermore, nondeterministic behavior confounds traditional debugging strategies, because a programmer cannot reliably expose buggy behavior over and over to zero in on a bug. Meanwhile, the programmer must account for several factors to optimize the performance of a Pthreaded program. Because creating a Pthread is an expensive operation, taking longer than 10,000 cycles, the programmer must amortize this overhead over a reasonably large amount of computation allocated to each Pthread. A typical strategy for minimizing this overhead is to create only as many Pthreads as there are processors on the system. Programmers must then strive to avoid serial bottlenecks and to balance the computation among these Pthreads. These complications are amplified when ***scalability*** is considered, that is, how quickly the program runs on different machines with different numbers of processor cores. Problems of nondeterminism and scalability are not unique to Pthreads, but persist in other threading models as well.

The nondeterminism of multithreaded programs has been viewed as a key reason that programming large-scale parallel applications remains error prone and difficult [235]. In response to the problem of nondeterminism, many researchers over multiple decades have advocated that the difficulty of parallel programming can be greatly reduced by using some form of deterministic parallelism [41, 42, 51, 60, 113, 114, 134, 158, 174, 187, 304, 310, 370, 414]. With a deterministic parallel program, the programmer observes no logical ***concurrency***, that is, no nondeterminacy in the behavior of the program due to the relative timing of communicating processes such as occurs when one process arrives at a lock before another.[6]

[6]Netzer and Miller [297] distinguish ***internal*** determinacy, where the program contains no determinacy races, from ***external*** determinacy, where the final answer of the program is the same from run to run. The work presented in this thesis generally seeks to avoid *programmer-observable* nondeterministic behavior.

The semantics of a deterministic parallel program therefore match those of a corresponding serial program, and reasoning about such a program's correctness, at least in theory, is no harder than reasoning about the correctness of a serial program. Testing, debugging, and formal verification are simplified, because there is no need to consider all possible relative interleavings of operations on shared mutable state. Furthermore, Blelloch *et al.* [52] argue persuasively that price of determinism in performance need not be high, and in particular, that deterministic parallel algorithms can be fast.

Despite the apparent advantages of deterministic parallelism, however, many parallel codes deployed in practice still exhibit nondeterministic behavior. For example, all the codes in the PARSEC [45], Galois [315], and STAMP [79] benchmark suites use concurrency mechanisms, such as mutex locks and condition variables, which behave nondeterministically. Part of the reason that many parallel programs are still nondeterministic is that existing parallel-programming environments are immature and do not provide good support for deterministic parallel programming. Indeed, concurrency mechanisms are the building blocks for synchronization in most parallel-programming environments. Programmers are familiar with locks and condition variables, and even generalizing them to transactional memory [182] does not mitigate their inherent nondeterminacy. Some approaches to determinism — such as data parallelism [224], commutative operations [331, 370], and various effect systems [60] — do not seem general enough to handle more than a limited set of real-life applications. Purely functional programming is perhaps the most straightforward way to achieve deterministic parallelism, because functional programming precludes parallel threads from interacting [51, 174], but few real-world codebases employ functional languages. Because of the prevalence of nondeterminism, only experts can program these parallel applications, especially those for shared-memory platforms, and these bug-prone codes can only be understood by experts.

Dynamic multithreading

Tremendous progress has been made in mitigating the problems posed by nondeterminism and scalability in parallel programming. Concurrency platforms that support ***dynamic multithreading***, such as dialects of Cilk [56, 106, 146, 196, 237, 246], Fortress [11], Habanero [33], Habanero-Java [82], Hood [59], HotSLAW [283], Java Fork/Join Framework [233], OpenMP [26, 305], Task Parallel Library [245], Threading Building Blocks (TBB) [330] and X10 [87], have been developed that relieve programmers of the need to explicitly deal with concurrency or load-balance a parallel program. Dynamic multithreading concurrency platforms typically support the programming model of ***fork-join parallelism***, in which subroutines can be spawned in parallel with little overhead. With dynamic multithreading, spawning a subroutine *allows*, but does not *require*, that subroutine to execute in parallel. The spawning of subroutines generates a series-parallel execution dag (directed acyclic graph) in which the synchronization of subtasks is managed "under the covers" by the runtime system. Constructs such as parallel loops can be implemented as syntactic sugar on top of the fork-join model. Dynamic multithreading concurrency platforms usually schedule the computation using ***randomized work-stealing*** [24, 56, 58, 146], where worker threads in the runtime system coordinate to load-balance the computation. Hence, dynamic multithreading provides a ***processor-oblivious*** model of parallel computation, in which programmers only expose logical parallelism without referring to the physical processor cores on the machine. Dynamic multithreading has grown in popularity to the point that mainstream compilers, such as GCC [369], ICC [198], and LLVM [232], now support the dynamic multithreading linguistics provided by Cilk Plus [151,195] and OpenMP [152,262].

Dynamic multithreading supports a principled approach to reasoning about parallel program correctness and scalability. Intuitively, as long as a parallel program contains no ***determinacy races*** [134] (also called ***general races*** [297]), then the program is deterministic, and it therefore exhibits serial semantics. Moreover, efficient tools exist that can guarantee to detect determinacy races or validate their absence [119,134,135,324,325,392]. The fork-join model also supports theoretical analysis of program performance and scalability in terms of the program's "work" and "span" [100, Ch. 27], and concurrency platforms can ensure that the performance predictions from work-span analysis are often borne out in practice [146,180]. Efficient tools exist for analyzing a program's scalability based work-span analysis [180].

Despite this progress, however, these technologies remain immature, and their shortcomings still leave programmers to deal with complex nondeterminism and scalability issues. How, for example, do dynamic multithreading platforms support efficient parallel computations on irregular structures (i.e., graphs), computations that use pseudorandom numbers, or computations that exhibit pipeline parallelism that emerges dynamically? How can programmers quickly see how different parts of a dynamic multithreaded program contribute to its "work" and "span?" How can programmers detect determinacy races in programs that use advanced programming features? How can compilers support performance engineering of dynamic multithreaded codes? The artifacts presented in this thesis shed light on these questions.

1.2 Developing simple programming models and theories of performance

The first five artifacts presented in this thesis (Chapters 3 through 7) explore the development of two key technologies to support a science of fast code: simple programming models and theories of performance that are borne out in practice. This section describes these two technologies as well as the five artifacts in this thesis that explore their development.

Simple programming models, such as the fork-join model of dynamic multithreading, can support principled approaches to reasoning about program behavior. Efficient multicore software, for example, must deal effectively with complex issues such as atomicity, nondeterminism, scheduling, and load-balancing. Programming models can simplify fast code by encapsulating "ugly" aspects of computer hardware and providing programmer-friendly abstract properties such as processor obliviousness, determinism, serial semantics, and composable performance. An analogy can be made with `goto` statements and the structured control-flow statements — e.g., conditionals and loops — that replaced them in modern serial programming languages. Similarly to how structured control-flow statements facilitates reasoning about the serial program behavior, simple programming models can make multicore programs easier to reason about than their traditional Pthreaded counterparts.

Some programming models support the powerful feature that they admit theories of performance that are borne out in practice, which support principled approaches to reasoning about software efficiency. Consider the randomized work-stealing schedulers [24,56,58,146] employed by dynamic multithreading concurrency platforms. These schedulers offer theoretical guarantees on how efficiently they execute a parallel program. By using these schedulers, programmers can predict how a parallel program will scale on any number of processors without having to write an optimized implementation and test it on a variety of machines and core counts. Moreover, programmers can perform back-of-the-envelope calculations based on work-span analysis to check whether parallel software meets its theoretical performance expectations or fails to do so due to a programming bug or an unanticipated factor that inhibits scalability. Because these theories of performance are borne out in practice, programmers can use the

The artifacts presented in this section address a variety of parallel programming challenges using dynamic multithreading, specifically, Cilk technology [56, 106, 146, 196, 237, 246]. Cilk supports a fork-join model of parallelism and employs a randomized work-stealing scheduler that schedules and load-balances programs efficiently, both in theory and in practice [146]. Modern dialects of Cilk [196, 237, 246] also support "reducer hyperobjects" [144], an advanced programming feature for managing concurrent accesses to shared variables.

The five artifacts overviewed in this section — "PBFS," "Prism," "Color," "DPRNG," and "Cilk-P" — explore how Cilk's programming model and theoretical performance guarantees can be employed and enhanced to support a wider variety of parallel computations. The "PBFS," "Prism," and "Color" artifacts examine how to parallelize algorithms on graphs to provide provable guarantees on efficiency that are borne out in practice. Graph computations are generally challenging to perform efficiently in parallel, because the irregular structure of a graph complicates the scheduling and load-balancing of these computations on parallel cores. The "DPRNG" artifact tackles the problem of generating pseudorandom numbers deterministically in parallel in dynamic multithreaded codes. The "Cilk-P" artifact studies the problem of writing and efficiently scheduling programs that exhibit both fork-join parallelism and pipeline parallelism that arises dynamically "on-the-fly."

PBFS

Chapter 3 presents PBFS [248], a "work-efficient" parallel breadth-first search (BFS) algorithm, which exhibits high work efficiency and achieves parallel speedup in both theory and practice.[7] In the standard serial BFS algorithm, the FIFO queue imposes a serial bottleneck that is difficult to parallelize effectively. PBFS replaces this queue with an efficient implementation of a multiset data structure, called a "bag," which allows PBFS to achieve parallel speedup in practice. On a variety of benchmark input graphs, PBFS runs as fast on a single processor as a tight implementation of the standard serial BFS algorithm, and it speeds up linearly with processors.

PBFS offers theoretical guarantees that support its performance in practice. PBFS is implemented in Cilk and employs Cilk's "reducer hyperobject" mechanism [144] to efficiently coordinate concurrent updates to bags. Because the Cilk runtime system maintains reducers based on how it schedules and load balances the computation, the work inherent in a PBFS execution is nondeterministic. My coauthor and I enhanced the dag model of Cilk dynamic multithreading to build a general theoretical performance model for analyzing Cilk programs that use reducers. Analyzing PBFS in this model shows that PBFS is ***work-efficient***, meaning that it performs asymptotically the same number of operations as its serial counterpart, the standard serial BFS algorithm. Furthermore, this analysis shows that PBFS achieves linear speedup on as many processors as the size of the graph divided by its diameter, up to polylogarithmic factors.

PBFS's theoretical guarantees offer assurance that the PBFS algorithm scales well when applied to other parallel software. Intel used PBFS to implement a parallel version of the

[7] I collaborated with Charles E. Leiserson on this work.

Murphi model checker [118] that achieves near-perfect parallel speedup, specifically, a 15.5 factor speedup on 16 cores.

Prism

Chapter 4 presents the Prism and Prism-R "chromatic-scheduling" algorithms for executing "dynamic data-graph computations" efficiently and deterministically in parallel [213].[8] A data-graph computation consist of rounds of updates, where each update recomputes the value associated with a vertex in a graph. A dynamic data-graph computation updates only a subset of the graph's vertices in each round. When executing a data-graph computation in parallel, locks are often employed to ensure that updates are performed atomically. Locks not only incur overhead, but they also cause updates to occur in a nondeterministic relative order.

Instead of locks, Prism uses "chromatic scheduling" [2, 43, 264] to coordinate parallel updates in a dynamic data-graph computation. Conceptually, Prism precomputes a vertex-coloring of the graph of dependencies between updates, and then executes each round of the data-graph computation based on the coloring: vertices of different colors are updated in a fixed order, while vertices of the same color are updated in parallel. Prism thereby ensures that the dynamic data-graph computation executes deterministically, allowing programmers to reason about the program's correctness based on its serial semantics. Prism employs an efficient implementation of a novel "multibag" data structure to maintain dynamic sets of vertices partitioned by color. Prism provides theoretical guarantees on its scalability, based on the size of the graph, the size of each round, and the number of colors used in the vertex-coloring. In practice, by avoiding the overheads of locking, Prism executes dynamic data-graph computations faster than lock-based alternatives, despite imposing the restrictive requirement of deterministic execution. By providing theoretical guarantees that are borne out in practice, programmers can rely on Prism to execute dynamic data-graph computations efficiently.

An extension of Prism, called Prism-R, handles dynamic data-graph computations whose updates perform associative operations on global variables. Prism-R employs an efficient implementation of a novel "multivector" data structure, in place of Prism's "multibag," to efficiently maintain ordered sets of vertices partitioned by color. Prism-R provides the same theoretical guarantees on its scalability to those of Prism, and runs nearly as quickly as Prism in practice.

Color

Chapter 5 presents the "largest-log-degree-first" (LLF) and "smallest-log-degree-last" (SLL) ordering heuristics for greedy graph-coloring algorithms [175].[9] These ordering heuristics seek to eliminate a tradeoff between coloring quality and parallel performance. Greedy graph-coloring algorithms often use ordering heuristics, such as "largest-degree-first" and "smallest-degree-last," to reduce the number of colors they use in practice. My coauthors and I show, however, that there exist input graphs on which these heuristics can create serial bottlenecks when applied to parallel graph-coloring algorithms. The LLF and SLL ordering heuristics aim to get the best of both worlds. Not only do these heuristics color graphs using

[8] Prism and Prism-R were developed in collaboration with Tim Kaler, William Hasenplaugh, and Charles E. Leiserson.

[9] I collaborated with William Hasenplaugh, Tim Kaler, and Charles E. Leiserson on this work.

a comparable number of colors to the analogous largest-degree-first and smallest-degree-last heuristics, but unlike their serial analogs, LLF and SLL provide theoretical guarantees on parallel scalability and achieve parallel speedup in practice. Programmers can therefore employ these heuristics in graph-coloring codes with confidence that they will not inhibit scalability, and they can use the theoretical guarantees to understand the performance of these graph coloring codes quantitatively.

DPRNG

Chapter 6 introduces the DotMix library and the "pedigree" mechanism to support deterministic parallel random-number generation in dynamic multithreaded programs [249].[10] To debug a serial program that uses pseudorandom numbers, programmers can fix the seed of the random-number generator to make it behave deterministically. Standard approaches to parallel random-number generation, however, produce pseudorandom numbers nondeterministically in dynamic multithreaded programs, even when the programmer uses a fixed seed. These approaches involve either synchronizing concurrent accesses to a common random-number generator or using separate random-number generators per processor. With either approach, the nondeterminism of parallel scheduling can change the pseudorandom numbers generated. Although some solutions to deterministic parallel random-number generation exist for Pthreaded programs [92,273,338], these solutions either do not scale or do not directly apply to dynamic multithreaded programs.

DotMix generates pseudorandom numbers in parallel, such that its behavior is deterministic for a fixed seed. The DotMix library thus simplifies the programming model for randomized dynamic multithreaded programs by providing linguistics to support serial semantics and repeatable execution. DotMix generates pseudorandom values of comparable statistical quality to the popular Mersenne twister random-number generator [274] while incurring little overhead — less than 21 % on realistic benchmarks — compared to using Mersenne twister nondeterministically. These properties ensure that programs that use DotMix still use high-quality pseudorandom numbers and pay minimal performance to enjoy deterministic behavior.

DotMix requires support from the runtime system and compiler to generate pseudorandom numbers deterministically. DotMix in particular uses our "pedigree" mechanism, which assigns a unique ID to each program point in a dynamic multithreaded program in a schedule-independent manner. To generate a pseudorandom number, each call to the `rand` routine in the DotMix library extracts the pedigree of the current program point and hashes it using a provably good hash function. This hash function guarantees a low probability that two ID's hash to the same value; formally, this hash function is 2-independent. Pedigrees incur negligible overhead in a runtime system, specifically, less than 1 % overhead in the MIT Cilk runtime system.

Cilk-P

Chapter 7 presents Cilk-P [239],[11] an extension to the Cilk concurrency platform that provides a simple, processor-oblivious programming model for writing deterministic parallel programs that exhibit "on-the-fly" pipeline parallelism. Pipeline parallelism [55, 157, 169, 268, 277, 295, 317, 328, 339, 381] is a well-known programming pattern that can be used to

[10] I collaborated with Charles E. Leiserson and Jim Sukha on this work.

[11] Cilk-P was joint work with I-Ting Angelina Lee, Charles E. Leiserson, Jim Sukha, and Zhunping Zhang.

parallelize a variety of applications, including streaming applications from the domains of video, audio, and digital signal processing. Many systems that support pipeline parallelism require the programmer to add locks and condition variables to application code in order to enforce such program dependencies that emerge on the fly, that is, during the program's dynamic execution. These synchronization mechanisms complicate the source code and necessarily behave nondeterministically.

Cilk-P introduces linguistic and runtime support for deterministic on-the-fly pipeline parallelism. Cilk-P's language constructs can encapsulate the nondeterminism of synchronization mechanisms needed to enforce pipeline dependencies, allowing programmers to write deterministic parallel programs. Furthermore, Cilk-P's language constructs are flexible, allowing the pipeline dependencies in the program to emerge on-the-fly, that is, as the program's execution unfolds dynamically. Cilk-P provides a simple programming model for pipeline parallelism that supports work-span analysis for measuring program scalability.

Cilk-P implements the PIPER scheduler to load-balance the program on parallel processors. PIPER provides theoretical guarantees to execute a pipeline program on any number of parallel processors using bounded space and nearly optimal time with high probability. Cilk-P's efficient implementation of PIPER ensures that these theoretical guarantees are borne out in practice. A Cilk-P implementation of the *x264* video encoder, for example, matches the performance of an optimized implementation in Pthreads that uses explicit synchronization mechanisms. Cilk-P thus allows programmers to write pipeline parallel programs simply and to reason about their scalability in a quantitative, scientific fashion.

1.3 Enhancing the software-development environment

The remaining four artifacts in this thesis (Chapters 8 through 11) focus on enhancing the software-development environment, specifically, diagnostic tools and the compiler, to support principled approaches to reasoning about program behavior and efficiency. Although simple programming models and theories of performance can support principled approaches to reasoning about program behavior and efficiency, tools and compiler technology can produce surprising or incomprehensible results if they do not assimilate these abstract models and theories. This section describes the service that diagnostic tools and compilers provide to developing efficient software. I overview the artifacts in this thesis that explore how abstract programming models and theories of performance can be embedded into diagnostic tools and the compiler. I also describe an artifact that tackles the complexity of developing new diagnostic tools.

Diagnostic tools can amplify a programmer's ability to reason about software that is otherwise too large or complex to understand, effectively redefining what counts as simple code. For example, provably effective determinacy-race detectors, such as those supported by Cilk [134,135,197,392], provide guarantees to verify the absence of a determinacy race or, if a race exists, to pinpoint its source. Furthermore, tools such as the Cilkview scalability analyzer [180] can measure the scalability of a Cilk program in terms of its "work" and "span." Tools have limited utility, however, is a programmer must wait a long time for them to produce results. Efficient diagnostic tools enable programmers to rapidly test new changes to their code and, thereby, to iteratively develop efficient software.

Compilers can relieve programmers of mechanical performance-engineering tasks that are onerous and error-prone to carry out manually. Compilers can automatically perform cumbersome transformations on software to improve its efficiency. These compiler optimizations allow programmers to write simpler codes that nevertheless execute efficiently. Compilers can also provide visibility into the program's dynamic execution through ***compiler instrumentation*** (e.g., [133, 350, 352]) — code inserted by a compiler into a program-under-test. Diagnostic tools can utilize compiler instrumentation to efficiently inspect the dynamic execution of a program, thus ensuring that programmers need not wait unduly for these tools to produce results.

To operate efficiently in accordance with the programmer's conceptual understanding of program behavior and performance, tools and compilers must incorporate the abstract programming and performance models that the programmer uses to reason about software. For example, to efficiently provide useful results, race detectors and scalability analyzers for Cilk programs leverage the Cilk programming model to quickly determine which instructions can execute in parallel. The Cilkview scalability analyzer, moreover, leverages the fact that the scalability of a Cilk program can be understood using work-span analysis. Meanwhile, mainstream compilers today often fail to perform valid optimizations across dynamic multithreading language constructs, which introduces unnecessary performance overhead to parallel codes in comparison to their serial counterparts.

The four artifacts presented in this section — "Cilkprof," "Rader," "Tapir," and "CSI" — tackle shortcomings in the software-development environment. The "Cilkprof," "Rader," and "Tapir" artifacts work to embed the abstract programming models and theories of performance supported by dynamic multithreading platforms into tools and compilers. "Cilkprof" takes on the problem of computing a profile of a program in terms of the program's "work" and "span." "Rader" studies the problem of finding programming bugs that yield determinacy races in Cilk programs that use "reducer hyperobjects" [144]. "Tapir" examines how existing compiler optimizations for serial code can be easily adapted to optimize dynamic multithreaded codes. The final artifact presented in this section, "CSI," takes on the complexity of developing new efficient diagnostic tools. In particular, "CSI" enables programmers to write efficient dynamic-analysis tools that use compiler instrumentation without having to modify the compiler.

Cilkprof

Chapter 8 presents the Cilkprof scalability profiler [346] for tracking down serial bottlenecks in a Cilk program.[12] When optimizing a serial program, a programmer can use a profiler, such as gprof [171], to measure how different parts of the program contribute to the program's overall running time. Programmers can use these measurements to methodically direct their optimization efforts towards expensive parts of the code. Although a programmer can use a standard profiler on a parallel program to measure how each processor spends its time, the resulting data do not necessarily tell the programmer what parts of the code prevent it from scaling onto additional processor cores.

Cilkprof measures how every call site — every program point that calls a function — affects the overall scalability of a Cilk program. In particular, Cilkprof build on the approach taken by Cilkview [180] to measure how each call site contributes to both the "work" and "span" of a given Cilk computation. With this information, programmers can work to improve a program's scalability by focusing their optimization efforts on functions that contribute substantially to the span. Cilkprof uses an efficient algorithm to accumulate measurements in an amortized "prof" data structure, which allows Cilkprof to compute its

[12] I collaborated with Bradley C. Kuszmaul, I-Ting Angelina Lee, Charles E. Leiserson, and William M. Leiserson on this work.

profile in time asymptotically equal to the serial running time of the program. In practice, Cilkprof computes profiles quickly, incurring only a constant-factor slowdown of 1.9 on average on a suite of application benchmarks. Programmers can use Cilkprof to discover serial bottlenecks in a Cilk program in a scientific fashion.

Rader

Chapter 9 presents Rader [240], a tool to detect race bugs that can cause an ostensibly deterministic Cilk program that uses "reducer hyperobjects" [144] to behave nondeterministically.[13] Although determinacy race detectors exist for Cilk programs, they cannot reliably detect race bugs that involve the state of a reducer, however, because the Cilk runtime system itself maintains this state nondeterministically.

Rader can detect race bugs that involve a Cilk reducer hyperobject. Rader implements the Peer-Set and SP+ race-detection algorithms, both of which provide guarantees to report a race if one exists. The Peer-Set algorithm evaluates a single execution of a Cilk program and guarantees to report a problem if the program ever reads the state of a reducer when the semantics of reducers deem it unsafe to do so. The SP+ algorithm, meanwhile, leverages the performance model described in Chapter 3 for analyzing Cilk programs that use reducers to detect race bugs that occur when the program accesses reducer state by means other than the reducer's API. The SP+ algorithm examines a particular execution of a Cilk program that uses reducers, which can be identified by the program, an input to the program, and a small "steal specification" that specifies the outcome of the runtime system's nondeterministic scheduling choices. These two algorithms enable programmers to methodically zero in on programming errors in a Cilk program that jeopardizes its deterministic behavior, enhancing their ability to contend with nondeterministic codes. We implemented both of these algorithms in Rader using compiler instrumentation on memory accesses, Cilk parallel language constructs, and operations on reducers. Both of these algorithms detect race bugs efficiently, both in theory and in practice.

Tapir

Chapter 10 presents the Tapir compiler intermediate representation (IR) [347], which enables ordinary compiler optimizations to effectively optimize dynamic multithreaded programs.[14] Mainstream compilers such as GCC [369], ICC [198], and LLVM [232] generally treat parallel language constructs as syntactic sugar for function calls into a parallel runtime system. These function calls impede valid optimizations on parallel code that the compiler readily performs on analogous serial code. It is well documented, however, that standard compiler optimizations for serial programs can introduce incorrect code when applied to parallel programs [282]. Intuitively, enabling mainstream compilers to optimize parallel code as effectively as it optimizes serial code requires extensive changes to their codebases to ensure that existing optimizations handle parallel code correctly.

Tapir embeds fork-join parallelism directly into LLVM IR to enable LLVM's existing optimizations for serial code to work on dynamic multithreaded programs with only minor changes. Tapir necessitates such minimal changes by representing fork-join parallel subroutines in an asymmetric fashion that captures the program's serial semantics. Implementing Tapir in LLVM thus required modifying only 5000 of LLVM's 3-million-line codebase. With

[13]Rader was joint work with I-Ting Angelina Lee.
[14]I collaborated with William Moses and Charles E. Leiserson on this work.

these changes, existing serial optimizations — such as common-subexpression elimination, loop-invariant code motion, and tail-recursion elimination — can operate across parallel control flow. Tapir also enables parallel optimizations such as "unnecessary-synchronization elimination" and "parallel-loop spawning." On a suite of 20 benchmark applications, my coauthors and I found that the applications exhibit higher work efficiency when compiled using our implementation of Tapir in LLVM than with other mainstream compilers. Tapir thus enables compilers to optimize dynamic multithreaded programs comparably to serial programs, ensuring that their theoretical performance guarantees are borne out in practice and allowing programmers to avail themselves of the powerful automatic optimization capabilities of a compiler when writing dynamic multithreaded programs.

Tapir programs also support provably good determinacy-race detection by race detectors similar to those that exist for dynamic multithreaded programs. We embed determinacy-race detection within the formal semantics for Tapir to formally prove that if an execution of a Tapir program is free of determinacy races, then it has serial semantics. This semantic feature simplifies the development of compiler optimizations for Tapir programs. In particular, this feature supports efficient verification that compiler optimizations do not introduce races into determinacy-race-free Tapir programs.

CSI

Chapter 11 presents the CSI compiler-instrumentation framework [345], which aims to lower the bar for developing new, efficient dynamic-analysis tools, including race detectors [123, 124, 134, 280, 281, 343, 351], memory checkers [31, 177, 350], cache simulators [121, 377, 407], call-graph generators [171, 205], code-coverage analyzers [390, 397], and performance profilers [171, 329, 401].[15] Although compiler instrumentation can provide visibility into the dynamic execution of a program, writing a dynamic-analysis tool that uses compiler instrumentation today generally requires the tool writer to modify the compiler to insert custom instrumentation for their tool. This requirement can raise the bar substantially for building new and innovative dynamic-analysis tools.

The CSI framework seeks to have the compiler insert comprehensive static instrumentation into a program-under-test. CSI's standard collection of instrumentation hooks aim to cover the instrumentation needs of a wide variety of dynamic-analysis tools. Each CSI-tool can be implemented simply as a library that defines relevant hooks; no further compiler modification is needed. Furthermore, the CSI runtime maintains information associated with each hook to simplify and speed up common tasks of dynamic-analysis tools, such as scanning over the "basic blocks" in a program or associating instrumentation with lines in the source code. CSI thereby enables programmers to rapidly develop new efficient diagnostic tools to scientifically investigate the dynamic execution of their programs.

Although CSI's approach to compiler instrumentation seems replete with overheads, CSI overcomes these overheads using modern compiler technology. In particular, unused hooks are elided during "link-time optimization (LTO)" [368], resulting in instrumented running times on par with custom instrumentation. A CSI port of Google's ThreadSanitizer race-detection tool [351], for example, exhibits performance approaching that of the original version that used custom compiler instrumentation. CSI furthermore preserves the performance of production code and does not slow down compilation and link times unduly. Compiling with CSI and linking with the "null" CSI-tool slows the build time of the Apache

[15]CSI was developed in collaboration with Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson.

HTTP server by less than 40 %, and the resulting tool-instrumented executable is as fast as the original uninstrumented code. CSI thus enables the easy development of diagnostic tools that are still efficient.

1.4 Thesis statement

The artifacts presented in this thesis develop software-performance-engineering technologies to support principled and scientific approaches to reasoning about efficient multicore software. Together, these artifacts provide evidence to support the following statement:

Thesis statement: We can develop a more coherent science of fast code for multicores by creating simple and integrated programming technologies that remedy the current *ad hoc* and unprincipled nature of software performance engineering.

These artifacts demonstrate three intertwined efforts to create such integrated technologies. Five of the artifacts build on dynamic multithreading technology to develop simple programming models that support theories of performance and scalability that are borne out in practice. Three artifacts work to embed the abstract programming models and theories of performance supported by dynamic multithreading into the diagnostic tools and compilers. The final artifact aims to enhance the software-development environment to facilitate the creation of efficient diagnostic tools. Together, these artifacts work to advance a science of fast code by supporting principled approaches to writing efficient software, reasoning about its behavior and performance, and carefully examining its dynamic execution.

I believe that, after the imminent end of Moore's Law, even average programmers will need to be able to write fast code in order to find further gains in computer performance. By building simple and integrated programming technologies to develop a coherent science of fast code and by educating programmers in that science, I believe that software performance engineering can be made accessible even to average programmers.

1.5 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents background on dynamic multithreading used throughout the remainder of this thesis. Chapters 3 through 11 present the nine artifacts contributed by this thesis:

- A work-efficient parallel breadth-first search algorithm, called PBFS, which exhibits high work efficiency and achieves parallel speedup in both theory and practice (Chapter 3).
- The pedigree mechanism and DotMix algorithm for generating pseudorandom numbers deterministically in parallel in dynamic multithreaded programs (Chapter 6).
- The Cilk-P concurrency platform, which provides linguistic and runtime support for deterministic on-the-fly pipeline parallelism (Chapter 7).
- A chromatic scheduling algorithm, called Prism, that executes dynamic data-graph computations deterministically and efficiently (Chapter 4).
- Ordering heuristics for parallel greedy graph coloring algorithms that provide provably good scalability that is borne out in practice (Chapter 5).
- Cilkprof, a profiler that efficiently measures how each call site in a Cilk program contributes to the program's scalability (Chapter 8).

- Rader, a provably good race detector for Cilk programs that use reducer hyperobjects (Chapter 9).
- The Tapir compiler intermediate representation, which enables existing compiler optimizations for serial code to optimize across parallel control flow with minimal changes (Chapter 10).
- CSI, a framework that provides comprehensive static instrumentation for dynamic-analysis tools (Chapter 11).

Chapter 12 concludes with a discussion on the end of Moore's Law and software performance engineering in the post-Moore era.

Chapter 2

Dynamic Multithreading

This chapter overviews dynamic multithreading, with a focus on the model supported by Cilk, which forms the basis of much of the work in this thesis. Cilk extends C/C++ with keywords for expressing logical fork-join parallelism within a program. The Cilk concurrency platform employs a randomized work-stealing scheduler in its runtime system to schedule and load-balance the computation on whatever processors are available at runtime. Modern dialects of Cilk also support a general parallel reduction mechanism, called "reducer hyper-objects." The code and pseudocode presented in subsequent chapters of this thesis shall use a dynamic multithreading model inspired by Cilk.

This chapter is organized as follows. Section 2.1 describes the **spawn**, **sync**, and **parallel for** language constructs for expressing fork-join parallelism. Section 2.2 describes how the execution of a dynamic multithreaded program can be modeled theoretically as a dag using the framework of Blumofe and Leiserson [58]. This section also overviews assumptions about the runtime environment. Section 2.3 defines deterministic computations. Section 2.4 describes work-span analysis of parallel scalability. Section 2.5 describes greedy and randomized work-stealing schedulers. Section 2.6 describes parallel reduction mechanisms and introduces "reducer hyperobjects."

2.1 Linguistic extensions for fork-join parallelism

Dynamic multithreading concurrency platforms typically support the abstraction of ***fork-join parallelism***, in which threads are spawned off as parallel subroutines. Although some platforms provide this abstraction through library routines, others provide it by extending a serial base language with language constructs. The dynamic multithreaded codes and pseudocodes presented in this thesis adopt the language approach. The pseudocodes in this thesis augment ordinary serial pseudocode with the keywords **spawn**, **sync**, and **parallel for**, of which **spawn** and **sync** are the more basic. Analogously, the code examples in this thesis are based on dialects of Cilk, most commonly Cilk Plus, which add the corresponding keywords `cilk_spawn`, `cilk_sync`, and `cilk_for` to ordinary C/C++ code.

Let us first examine the dynamic multithreading pseudocode keywords **spawn** and **sync**. Parallel work is created when the keyword **spawn** precedes the invocation of a function, thereby causing the function to be ***spawned***. Semantically, spawning a function differs from calling a function only in that the parent ***continuation*** — the code immediately following the spawn — is allowed to execute in parallel with the invoked child function, instead of waiting for the child to complete, as is normally done for a function call. A function cannot

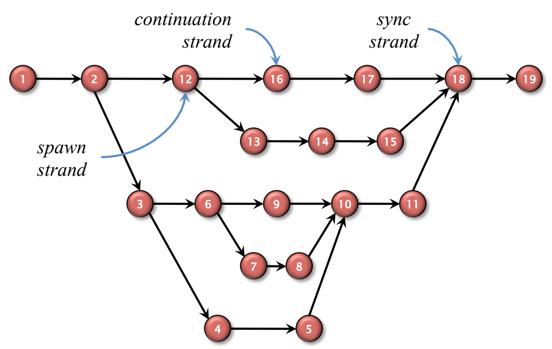


Figure 2-1: A dag representation of the execution of a dynamic multithreaded program. Each vertex represents a strand, and edges represent parallel-control dependencies between strands.

safely use the values returned by its spawned children until it executes a **sync** statement, which suspends the function until all of its spawned children return. The function is ***synced*** once its spawned children return and it is allowed to execute the instruction after the **sync**. Every function syncs implicitly before it returns, precluding orphaning. Together, **spawn** and **sync** can succinctly express fork-join parallelism in a program. The scheduler in the runtime system takes the responsibility of scheduling the spawned functions on the individual processor cores of the multicore computer and synchronizing their returns according to the fork-join logic provided by the **spawn** and **sync** keywords.

Loops can be parallelized by preceding an ordinary **for** with the keyword **parallel**, which indicates that all iterations of the loop may operate in parallel. Parallel loops do not require additional runtime support, but can be implemented by parallel divide-and-conquer recursion over the loop iterations using **spawn** and **sync**.

A key property of this linguistic model is ***processor-obliviousness***, meaning that a dynamic-multithreaded program makes no explicit reference to the processors on which it executes. As a consequence, a dynamic multithreaded program admits a serial execution. Simply eliding the keywords **spawn**, **sync**, and **parallel** produces a serial program, called the ***serialization*** or ***serial elision***, which implements a legal semantics of the dynamic multithreaded program. The serialization has the property that spawned children are simply called, and they complete their execution before the parent resumes, as with an ordinary function call.

2.2 The dag model

The dag model views the executed computation resulting from running a dynamic multithreaded program[1] as a ***dag (directed acyclic graph)***, where a vertex represents a ***strand*** — a sequence of serially executed instructions containing no parallel control — and an edge (u, v) represents a (parallel-)control dependency between two strands u and v which asserts that v cannot execute until after u has finished executing. Figure 2-1 illustrates such a dag, which notably involves executed instructions, as opposed to source instructions. A strand can be as small as a single instruction, or it can represent a longer computation. Generally, we shall dice a chain of serially executed instructions into strands in a manner that is convenient for the computation we are modeling. For notational convenience, we shall typically assume that strands respect function boundaries, meaning that calling or spawning

[1]When we refer to the running of a program, we shall generally assume that we mean "on a given input."

a function terminates a strand, as does returning from a function. Each strand therefore belongs to exactly one function instantiation.

We identify a few types of strands in the dag modeling an executed computation. A strand that has 2 outgoing control dependencies is a ***spawn strand***, and a strand that resumes the caller after a spawn is called a ***continuation strand***. A strand that has at least 2 incoming control dependencies is a ***sync strand***. We assume that no continuation strand is also a sync strand.

The serial execution of a dynamic multithreaded program corresponds to a walk of its dag. Because we assume that a serial execution of the computation executes all spawned child subcomputations before their continuations, the ***serial (execution) order*** of a computation corresponds to a depth-first traversal of the dag in which the spawned child of a spawn strand is executed before its continuation.

The ***length*** of a strand is the time it takes for a processor to execute all its instructions. For most theoretical analyses in this paper, we shall assume that programs execute on an ***ideal parallel computer***, where each instruction takes unit time to execute, there is ample memory bandwidth, there are no cache effects, etc. We shall also assume that the computer supports concurrent reads and writes.

2.3 Determinacy and races

We say that a dynamic multithreaded program is ***deterministic*** (on a given input) if every memory location is updated with the same sequence of values in every execution. Otherwise, the program is ***nondeterministic***. A deterministic program always behaves the same, no matter how the program is scheduled. Two different memory locations might be updated in different orders, but each location always sees the same sequence of updates. Whereas a nondeterministic program may produce different dags, i.e., behave differently, a deterministic program always produces the same dag.

A program execution with no "determinacy races" is deterministic and always produces the same dag, no matter how the execution is scheduled. A program execution contains a ***determinacy race*** [134] if two parallel strands access the same memory location and at least one of the strands updates it. Because the serialization of a dynamic multithreaded program implements legal semantics for the program, every execution of a determinacy-race-free program has the same semantics as its serialization. In this case, we say that a determinacy-race-free dynamic multithreaded program has ***serial semantics***.

Determinacy races have been given many different names in the literature. For example, they are sometimes called ***access anomalies*** [122], ***data races*** [280], ***race conditions*** [370], or ***harmful shared-memory accesses*** [302]. Netzer and Miller [297] clarify different types of races and define a determinacy race or ***general race*** to be a race that causes a supposedly deterministic program to behave nondeterministically. (They also define a ***data race*** or ***atomicity race*** to be a race in a nondeterministic program involving nonatomic accesses to critical regions.) We shall prefer the more descriptive term "determinacy race." Emrath and Padua [130] call a deterministic program ***internally deterministic*** if the program execution on the given input exhibits no determinacy race and ***externally deterministic*** if the program has determinacy races but its output is deterministic because of the commutative and associative operations performed on the shared locations.

2.4 Work-span analysis

The dag model admits two natural measures of the performance of a dynamic multithreaded computation. The ***work*** of a dag A, denoted by Work(A), is the sum of the lengths of all the strands in the dag. Assuming for simplicity that it takes unit time to execute each strand, the work for the example dag in Figure 2-1 is 19. The ***span***[2] of A, denoted by Span(A), is the length of the longest path in the dag. Again assuming unit-time strands, the span of the dag in Figure 2-1 is 10, which is realized by the path $\langle 1, 2, 3, 6, 7, 8, 10, 11, 18, 19\rangle$. Work-span analysis is outlined in tutorial fashion in [100, Ch. 27] and in [246].

Work and span can be used to provide important bounds [57, 58, 66, 128, 170] on performance and speedup. Suppose that a program execution produces a dag A in time T_P when run on P processors of an ideal parallel computer. We have the following two lower bounds on the execution time T_P:

$$T_P \geq \text{Work}(A)/P \; , \tag{2.1}$$

$$T_P \geq \text{Span}(A) \; . \tag{2.2}$$

Inequality (2.2), which is called the ***Work Law***, holds in this simple performance model, because each processor executes at most 1 instruction per unit time, and hence P processors can execute at most P instructions per unit time. Inequality (2.2), called the ***Span Law***, holds because no execution that respects the partial order of the dag can execute faster than the longest serial chain of instructions.

We define the ***speedup*** of a program as T_1/T_P — how much faster the P-processor execution is than the serial execution. For deterministic programs, since all executions produce the same dag A, we have that $T_1 = \text{Work}(A)$, and $T_\infty = \text{Span}(A)$ (assuming no overhead for scheduling). Rewriting the Work Law, we obtain $T_1/T_P \leq P$, which is to say that the speedup on P processors can be at most P. If the application obtains speedup P, which is the best we can do in our model, we say that the application exhibits ***linear speedup***. If the application obtains speedup greater than P (which cannot happen in our model due to the Work Law, but can happen in models that incorporate caching and other processor effects), we say that the application exhibits ***superlinear speedup***.

The ***parallelism*** of the dag is defined as Work(A)/Span(A). For a deterministic computation, the parallelism is therefore T_1/T_∞. The parallelism represents the maximum possible speedup on any number of processors, which follows from the Span Law, because $T_1/T_P \leq T_1/\text{Span}(A) = \text{Work}(A)/\text{Span}(A)$. For example, the parallelism of the dag in Figure 2-1 is $19/10 = 1.9$, which means that any advantage gained by executing it with more than 2 processors is marginal, since the additional processors will surely be starved for work. The parallelism simultaneously provides a limit on the possibility of attaining perfect linear speedup. To see this point, suppose that $P > \text{Work}(A)/\text{Span}(A)$. In this case the span law implies that the speedup satisfies $T_1/T_P \leq \text{Work}(A)/\text{Span}(A) < P$. Moreover, in an ideal parallel computer, the more processors that are used beyond the parallelism — the more that P exceeds Work(A)/Span(A) — the less perfect the parallel speedup.

In practice, to achieve linear speedup and minimize the overhead of Cilk's randomized work-stealing scheduler, a Cilk computation should exhibit ample parallelism, that is, the parallelism of the computation should exceed the number of processors by a sufficient margin [146], typically a factor of 10 [100, p. 783].

[2] The literature also uses the terms ***(computational) depth*** [51] and ***critical-path length*** [56].

2.5 Scheduling

A ***greedy scheduler*** [57,66,128,170] schedules a computation without ever leaving a processor idle if there is work that can be done. If a computation scheduled by a greedy scheduler produces a dag A, then we have

$$T_P \leq \text{Work}(A)/P + \text{Span}(A) \; . \tag{2.3}$$

This bound assumes an ideal computer and ignores overheads for scheduling. For a deterministic computation, if the parallelism exceeds the number P of processors by a sufficient margin, Inequality (2.3) guarantees near-perfect linear speedup. Specifically, if $P \ll \text{Work}(A)/\text{Span}(A)$, then $\text{Span}(A) \ll \text{Work}(A)/P$, and hence Inequality (2.3) yields $T_P \approx \text{Work}(A)/P$, and the speedup is $T_1/T_P \approx P$.

A ***(randomized) work-stealing*** scheduler [24,56,58,146] operates as follows. When the runtime system starts up, it allocates as many operating-system threads, called ***workers***, as there are processors (although the programmer can override this default decision). Each worker's stack operates like a ***deque***, or double-ended queue. When a subroutine is spawned, the subroutine's activation frame containing its local variables is pushed onto the bottom of the deque. When it returns, the frame is popped off the bottom. Thus, in the common case, the parallel code operates just like serial code and imposes little overhead. When a worker runs out of work, however, it becomes a ***thief*** and "steals" the top frame from another ***victim*** worker's deque, where the victim worker is chosen uniformly at random from the set of workers. In general, the worker operates on the bottom of the deque, and thieves steal from the top. This strategy has the great advantage that all communication and synchronization is incurred only when a worker runs out of work. If an application exhibits sufficient parallelism, stealing is infrequent, and thus the cost of bookkeeping, communication, and synchronization to effect a steal is negligible.

Work-stealing achieves good expected running time based on the work and span. In particular, if A is the executed dag on P processors, the expected execution time T_P can be bounded as

$$T_P \leq \text{Work}(A)/P + O(\text{Span}(A)) \; , \tag{2.4}$$

where we omit the notation for expectation for simplicity. This bound, which is proved in [58], assumes an ideal computer, but it includes scheduling overhead. For a deterministic computation, if the parallelism exceeds the number P of processors sufficiently, Inequality (2.4) guarantees near-linear speedup. Specifically, if $P \ll \text{Work}(A)/\text{Span}(A)$, then $\text{Span}(A) \ll \text{Work}(A)/P$, and hence Inequality (2.4) yields $T_P \approx \text{Work}(A)/P$, and the speedup is $T_1/T_P \approx P$.

Another relevant measure is the number of steals that occur during a computation. As is shown in [58], the expected number of steals incurred for a dag A produced by a P-processor execution is $O(P \cdot \text{Span}(A))$.

2.6 Parallel reduction mechanisms

A popular technique for coordinating safe concurrent updates to a shared variable or data structure without contention is to use ***parallel reductions*** [50, 86, 204, 224, 231, 278, 330]. Many modern concurrency platforms provide some form of (parallel) reduction mechanism [144, 196, 241, 278, 306, 330, 356, 409]. A reduction mechanism coordinates parallel up-

(a)

```
1   $x = 10$
2   $x{+}{+}$
3   $x \mathrel{+}= 3$
4   $x \mathrel{+}= -2$
5   $x \mathrel{+}= 6$
6   $x{-}{-}$
7   $x \mathrel{+}= 4$
8   $x \mathrel{+}= 3$
9   $x{+}{+}$
10  $x \mathrel{+}= -9$
```

(b)

```
1   $x = 10$
2   $x{+}{+}$
3   $x \mathrel{+}= 3$
4   $x \mathrel{+}= -2$
5   $x \mathrel{+}= 6$
    $x' = 0$
6   $x'{-}{-}$
7   $x' \mathrel{+}= 4$
8   $x' \mathrel{+}= 3$
9   $x'{+}{+}$
10  $x' \mathrel{+}= -9$
    $x \mathrel{+}= x'$
```

(c)

```
1   $x = 10$
2   $x{+}{+}$
3   $x \mathrel{+}= 3$
    $x' = 0$
4   $x' \mathrel{+}= -2$
5   $x' \mathrel{+}= 6$
6   $x'{-}{-}$
    $x'' = 0$
7   $x'' \mathrel{+}= 4$
8   $x'' \mathrel{+}= 3$
9   $x''{+}{+}$
10  $x'' \mathrel{+}= -9$
    $x \mathrel{+}= x'$
    $x \mathrel{+}= x''$
```

Figure 2-2: The intuition behind reducers. **(a)** A series of additive updates performed on a variable x. **(b)** The same series of additive updates split between two "views" x and x'. The two update sequences can execute in parallel and are combined at the end. **(c)** Another valid splitting of these updates among the views x, x', and x''.

dates to a shared variable by accumulating concurrent updates in distinct, local ***views*** of the variable. When the parallel subcomputations that update the variable complete, these views are combined together, or ***reduced***, using a binary REDUCE operator. A reduction mechanism typically encapsulates the nondeterministic behavior induced by parallel updates as long as the update and reduce operations satisfy associativity and commutativity.

Modern dialects of Cilk, including Cilk++ [246], Cilk Plus [196], and Cilk-M [238], provide a parallel reduction mechanism called a ***reducer hyperobject*** [144] (or simply a ***reducer***). A reducer is defined in terms of a binary associative REDUCE operator, such as sum, list concatenation, logical AND, etc. Like other parallel reduction mechanisms, updates to a reducer are accumulated in local ***views***, which the runtime system combines automatically with calls to REDUCE when subcomputations join. Unlike other reduction mechanisms, however, the REDUCE operator need not be commutative; as long as REDUCE is associative, then the Cilk program has serial semantics.

Figure 2-2 illustrates the basic idea of a reducer. The example involves a series of additive updates to a variable x. When the code in Figure 2-2(a) is executed serially, the resulting value is $x = 16$. Figure 2-2(b) shows the same series of updates split between two "views," x and x', of the variable. These two views may be evaluated independently in parallel and then combined, or ***reduced***, at the end, as Figure 2-2(b) shows. As long as the values for the views x and x' are not inspected in the middle of the computation, the associativity of addition guarantees that the final result is deterministically $x = 16$. This series of updates could be split anywhere else along the way and yield the same final result. Figure 2-2(c) demonstrates, for example, how the computation can be split across three views, x, x', and x''. To encapsulate nondeterminism in this way, each of the views must be reduced with an associative REDUCE operator (addition for this example) and intermediate views must be initialized to the identity for REDUCE (0 for this example).

Reducer hyperobjects support this kind of decomposition of update sequences automatically without requiring the programmer to manually create various views. When a function spawns, the spawned child inherits the parent's view of the reducer. If the child returns before the continuation executes, the child can return the view and the chain of updates can continue. If the continuation begins executing before the child returns, however, then the continuation receives a new view initialized to the identity for the associative REDUCE operator. Sometime at or before the **sync** that joins the spawned child with its parent, the two views are combined with REDUCE. If REDUCE is indeed associative, then the result is the same as if all the updates had occurred serially. Indeed, if the program is run on one processor, then the entire computation updates only a single view without ever invoking the REDUCE operator, in which case the behavior is virtually identical to a serial execution that uses an ordinary object instead of a hyperobject.

Formally, a reducer is defined in terms of an algebraic ***monoid***: a triple $(T, \otimes, e)$, where T is a set and $\otimes$ is an associative binary operation over T with identity e. From an object-oriented programming perspective, the set T is a base type which provides a member function REDUCE that implements the binary operator $\otimes$ and a member function CREATE-IDENTITY that constructs an identity element of type T. The base type T also provides one or more UPDATE functions, which modify an object of type T. The reducer library included in modern Cilk dialects provides a list of commonly used monoids. A programmer can also declare a reducer with a user-defined view type, so long as she defines a CREATE-IDENTITY function to create an identity view of that type and a REDUCE function that implements a binary associative operator for that type.

Reducers complicate the ordinary work-span analysis of a Cilk program, because the Cilk runtime system maintains views of a reducer nondeterministically. Chapter 3 describes this nondeterminism and introduces a performance model for bounding the parallel execution time of a program that uses reducers. In particular, Chapter 3 shows that Cilk programs that use reducers achieve the expected runtime bound:

$$T_P \le \text{Work}(A_\nu)/P + O(\tau^2 \cdot \text{Span}(A_\nu)) \ , \tag{2.5}$$

where A_ν is the "user dag" of A — the dag from the programmer's perspective — and τ is an upper bound on the time it takes to perform a REDUCE, which may be a function of the input size. For nondeterministic computations satisfying Inequality (2.5), we can define the ***effective parallelism*** as $\text{Work}(A_\nu)/(\tau^2 \cdot \text{Span}(A_\nu))$. Just as with parallelism for deterministic computations, if the effective parallelism exceeds the number P of processors by a sufficient margin, then the P-processor execution is guaranteed to attain near-linear speedup over the serial execution.

2.7 Worker-local storage

Worker-local storage[3] is memory that is private to a particular worker. Conceptually, in a P-processor execution of a parallel program, a worker-local variable can be implemented using a shared-memory array of P instances of that variable, where each instance belongs to a particular worker. Work-stealing runtime systems frequently employ worker-local storage to store the state of each worker in the system, such as the worker's deque.

In practice, many dynamic-multithreading concurrency platforms offer a mechanism to support worker-local storage in a user-level program. The Intel Cilk Plus runtime system, for example, provides the `__cilkrts_get_worker_number` API call, which returns an integer that identifies the worker executing a strand, and the `__cilkrts_get_nworkers` API call, to get P, the number of workers in the system. A Cilk program can therefore implement

[3]Also called ***thread-local storage*** [372].

its own variable x in worker-local storage by allocating its own array of P instances of x and allowing each strand to access the instance of x corresponding to the ID of the worker executing the strand.

Although worker-local storage shares some similarities with reducers — both mechanisms, for example, can ensure that distinct workers see distinct instances of a variable — worker-local storage differs from reducers in several important ways. Variables in a dynamic multithreaded program that are stored in worker-local storage can behave nondeterministically, based on the runtime system's nondeterministic choice of which worker executes which strand. A reducer, meanwhile, provides a processor-oblivious abstraction for a variable and adheres to "peer-set semantics" (defined in Chapter 9) which dictates when reading the value of the reducer is guaranteed to return a deterministic result. Furthermore, multiple simultaneous runs of a function that uses worker-local variables can interfere if the programmer is not careful to ensure that the different function instantiations use disjoint regions of worker-local storage. Chapter 3 compares worker-local storage and reducers in more detail. Of the dynamic multithreaded algorithms described in this thesis, only Prism and Prismr (Chapter 4) use worker-local storage.

Chapter 3

A Work-Efficient Parallel Breadth-First Search Algorithm

This chapter examines the problem of conducting a breadth-first search of a graph in parallel and presents PBFS [248, 344], a work-efficient algorithm to solve this problem. This work was conducted in collaboration with Charles E. Leiserson.

3.1 Introduction

Algorithms to search a graph in a breadth-first manner have been studied for over 50 years. The first breadth-first search (BFS) algorithm was discovered by Moore [286] while studying the problem of finding paths through mazes. Lee [234] independently discovered the same algorithm in the context of routing wires on circuit boards. A variety of parallel BFS algorithms have since been explored [29, 36, 96, 225, 413, 418]. Some of these parallel algorithms are ***work efficient***, meaning that the total number of operations performed is the same to within a constant factor as that of a comparable serial algorithm. That constant factor, which we call the ***work efficiency***, can be important in practice, but few if any papers actually measure work efficiency. We present a parallel BFS algorithm, called PBFS, whose performance scales linearly with the number of processors, both in theory and in practice, and for which the work efficiency is nearly 1, as measured by comparing its performance on benchmark graphs to the classical FIFO-queue algorithm [100, Section 22.2].

Given a graph $G = (V, E)$ with vertex set $V = V(G)$ and edge set $E = E(G)$, the ***BFS problem*** is to compute for each vertex $v \in V$ the distance $v.\mathit{dist}$ that v lies from a distinguished ***source*** vertex $v_0 \in V$. The BFS problem measures distance as the minimum number of edges on a path from v_0 to v in G. To simplify the statement of results, we shall assume that G is connected and undirected, although the algorithms we shall explore apply equally as well to unconnected graphs, digraphs, and multigraphs.

Figure 3-1 presents SERIAL-BFS, a variant of the classical serial algorithm [100, Section 22.2] for computing BFS which uses a FIFO queue as an auxiliary data structure. The FIFO can be implemented simply as an array with two pointers to the head and tail of the items in the queue. Enqueueing an item consists of incrementing the tail pointer and storing the item into the array at the new pointer location. Dequeueing consists of removing the item referenced by the head pointer and incrementing the head pointer. Because only $\Theta(1)$ operations are required to enqueue or dequeue an item, the work of SERIAL-BFS is $\Theta(V + E)$. Moreover, the constants hidden by the asymptotic notation are small due to the

```
SERIAL-BFS($G, v_0$)
1   for $u \in V(G) - \{v_0\}$
2       $u.dist = \infty$
3   $v_0.dist = 0$
4   $Q = \{v_0\}$
5   while $Q \neq \emptyset$
6       $u = $ DEQUEUE($Q$)
7       for $v \in V(G)$ such that $(u, v) \in E(G)$
8           if $v.dist == \infty$
9               $v.dist = u.dist + 1$
10              ENQUEUE($Q, v$)
```

Figure 3-1: A standard serial breadth-first search algorithm operating on a graph G and source vertex $v_0 \in V(G)$. The algorithm employs a FIFO queue Q as an auxiliary data structure to compute for each $v \in V(G)$ the distance $v.dist$ that v lies from v_0.

extreme simplicity of the FIFO operations.

Although efficient, the FIFO queue hinders parallelization of SERIAL-BFS. Parallelizing SERIAL-BFS while leaving the FIFO queue intact yields minimal parallelism for ***sparse*** graphs — graphs for which $|E| \approx |V|$. The reason is that if each ENQUEUE operation must be serialized, then the span of the computation must have length $\Omega(V)$. A work-efficient algorithm can thus have parallelism at most $\Theta(V + E)/\Omega(V)$, which is $O(1)$ when $|E| = O(V)$.[1]

Replacing the FIFO queue with another data structure can compromise the work-efficiency of a BFS algorithm, however, because FIFO's are so simple and fast. This work introduces a multiset data structure, called a ***bag***, which supports insertion essentially as fast as a FIFO, even when constant factors are considered. In addition, bags can be split and unioned efficiently.

We have designed a parallel BFS algorithm and implemented it in Cilk++ [190, 246]. The ***PBFS*** algorithm, which employs bags instead of a FIFO, uses the reducer hyperobject [144] feature of Cilk++. Our implementation of PBFS runs comparably on a single processor to a good implementation of SERIAL-BFS. The parallelism of PBFS on a given graph G decreases with the ***diameter*** of G, that is, the maximum distance between any two vertices in G. For a variety of benchmark graphs whose diameters are significantly smaller than the number of vertices — a common occurrence in practice — PBFS demonstrates high levels of parallelism and generally good speedup with the number of processor cores.

Figure 3-2 shows the typical speedup our PBFS implementation obtains on a large benchmark graph, in this case, for a sparse matrix called Cage15 arising from DNA electrophoresis [395]. This undirected graph has $|V| = 5{,}154{,}859$ vertices and $|E| = 99{,}199{,}551$ edges, and a BFS finds all vertices within distance 50 of the source vertex. The code was run on an Intel Core i7 machine with eight 2.53 GHz processing cores, 12 GiB of RAM, and two 8 MiB L3-caches, each shared among 4 cores. As the figure shows, although PBFS scales well initially, it attains a speedup of only about 5 on 8 cores, even though the parallelism in this graph is nearly 700. The figure additionally plots the impact of artificially increasing the ***computational intensity*** — the ratio of the number of CPU operations to the number of memory operations — and suggests that this low speedup is due to limitations of the memory system, rather than to the inherent parallelism in the algorithm.

PBFS exhibits two sources of nondeterminism. First, because the program employs a

[1]For convenience, the notation for set cardinality is omitted within asymptotic notation.

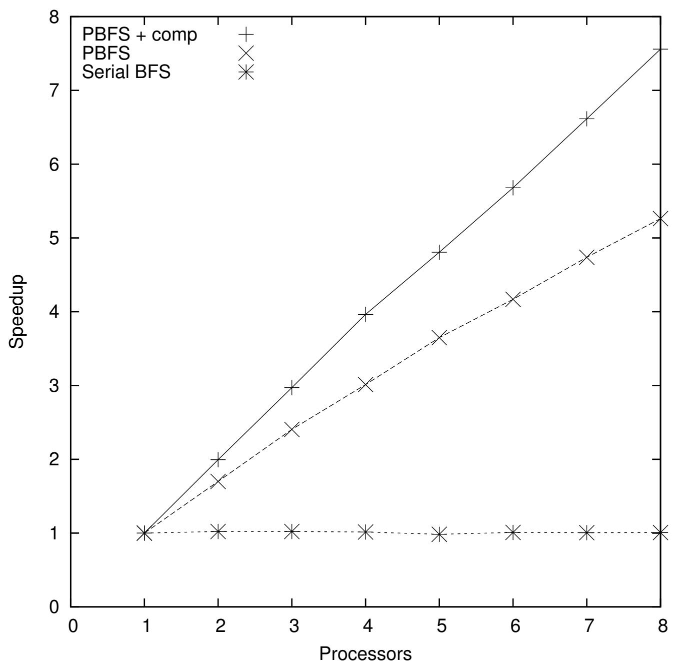


Figure 3-2: The performance of PBFS for the Cage15 graph showing speedup curves for serial BFS, PBFS, and a variant of PBFS where the computational intensity has been artificially enhanced and the speedup normalized.

bag reducer which operates in nonconstant time, the work and span of PBFS can vary from run to run depending upon how Cilk++'s work-stealing scheduler load-balances the computation. Second, for efficient implementation, PBFS contains a benign race condition, which does not affect the correctness of PBFS, but can cause PBFS to perform additional work nondeterministically.

Our theoretical analysis of PBFS bounds the additional work due to the bag reducer when the race condition is resolved using mutual-exclusion locks. For example, on a connected graph G with vertex set $V = V(G)$, edge set $E = E(G)$, diameter D, and $O(1)$ maximum ***out-degree*** — maximum number of outgoing edges from any vertex $v \in V$ — this "locking" version of PBFS performs a BFS in $\Theta(V+E)/P+O(D\lg^3(V/D))$ time on P processors and exhibits effective parallelism $\Omega((V+E)/D\lg^3(V/D))$, which is considerable when $D \ll V$, even if G is sparse. Our method of analysis is general and can be applied to other programs that employ reducers. Consequently, this model allows programmers to predict how the use of a reducer will affect the performance of any Cilk program. We leave it as an open question how to analyze the extra work when the race condition is left unresolved.

PBFS's theoretical guarantees justify the scalability that our PBFS implementation displays. Consequently, when applying PBFS to other software, performance engineers can make quantitative predictions about how PBFS will perform and have confidence that PBFS will satisfy those predictions in practice. Section 3.4 describes an example application of PBFS to performance-engineering another program. Dr. Yuxiong He, formerly of Cilk Arts and Intel Corporation, used PBFS to parallelize the Murphi model-checking tool [118] and observed that the parallel Murphi using PBFS scales well, attaining a large speedup factor of 15.5 on 16 cores.

Outline

The main body of this chapter first describes PBFS and its empirical performance and then delves into its theoretical analysis.

Sections 3.2 through 3.4 describe PBFS and its empirical performance. Section 3.2 describes the basic PBFS algorithm, and Section 3.3 describes the implementation of the bag data structure. Section 3.4 studies PBFS's empirical performance.

Sections 3.5 through 3.7 describe how to cope with the nondeterminism of reducers in the theoretical analysis of PBFS. Section 3.5 gives a formal model for reducer behavior, and Section 3.6 develops a theory for analyzing programs that use reducers. Section 3.7 employs this theory to analyze the performance of PBFS.

Section 3.8 concludes by discussing thread-local storage as an alternative to reducers.

3.2 The PBFS algorithm

This section presents the PBFS algorithm. We describe the "layer synchronization" strategy PBFS employs, which requires PBFS to use a data structure to maintain a "layer" of the graph. We describe how PBFS uses a "bag" to implement a layer and perform breadth-first search. Section 3.3 describes the "bag" data structure in detail.

PBFS uses ***layer synchronization*** [29, 418] to parallelize the breadth-first search of an input graph G. Let $v_0 \in V(G)$ be the source vertex, and define ***layer*** d to be the set $V_d \subseteq V(G)$ of vertices at distance d from v_0. For example, layer 0 is $V_0 = \{v_0\}$. Each iteration processes layer d by checking all the neighbors of vertices in V_d for those that should be added to V_{d+1}.

PBFS implements layers using an unordered-set data structure, called a ***bag***, which supports efficient parallel traversal over the elements in the set and provides the following operations:

- S = BAG-CREATE(): Create a new empty bag.
- BAG-INSERT(S, x): Insert element x into *bag*.
- BAG-UNION(S_1, S_2): Move all the elements from S_2 into S_1, and destroy S_2.

As Section 3.3 shows, BAG-CREATE operates in $O(1)$ time, and BAG-INSERT operates in $O(1)$ amortized time and $O(\lg n)$ worst-case time on a bag with n elements. Moreover, BAG-UNION operates in $O(\lg n)$ worst-case time.

Let us walk through the pseudocode for PBFS, which is shown in Figure 3-3. For the moment, ignore the **revert** and **reducer** keywords in lines 18 and 19.

After initialization, PBFS begins the **while** loop in line 17 which iteratively calls the auxiliary function PROCESS-LAYER to process layer $d = 0, 1, \ldots, D$, where D is the diameter of the input graph G. Section 3.3 walks through the pseudocode of PROCESS-LAYER and PROCESS-PENNANT in detail, but we shall give a high-level description of these functions here. To process V_d, PROCESS-LAYER extracts each vertex u in V_d in parallel and examines each edge (u, v) in parallel. If v has not yet been visited — $v.dist$ is infinite (line 27) — then line 28 sets $v.dist = d + 1$ and line 29 inserts v into bag V_{d+1}.

This description skirts over two subtleties that require discussion, both involving races.

First, the update of $v.dist$ in line 28 creates a race, since a processor processing vertex u and one processing vertex u' might both be examining vertex v at the same time. Theoretically, both processors might check whether $v.dist$ is infinite in line 27, discover that it is, and both proceed to update $v.dist$. Fortunately, this race is benign, meaning that it does not affect the correctness of the algorithm. Both processors set $v.dist$ to the same value, and

```
PBFS($G, v_0$)
11  parallel for $v \in V(G) - \{v_0\}$
12      $v.dist = \infty$
13  $v_0.dist = 0$
14  $d = 0$
15  $V_0$ = BAG-CREATE()
16  BAG-INSERT($V_0, v_0$)
17  while $V_d \neq \emptyset$
18      $V_{d+1}$ = new reducer BAG-CREATE()
19      PROCESS-LAYER(revert $V_d, V_{d+1}, d$)
20      $d = d + 1$

PROCESS-LAYER($V_d, V_{d+1}, d$)
21  for $k = \lfloor \lg |V_d| \rfloor$ downto 0
22      if $V_d[k] \neq$ NULL
23          PROCESS-PENNANT($V_d[k], V_{d+1}, d$)

PROCESS-PENNANT($N, V_{d+1}, d$)
24  if PENNANT-SIZE($N$) < GRAINSIZE
25      for $u \in N$
26          parallel for $v \in \text{Adj}[u]$
27              if $v.dist == \infty$
28                  $v.dist = d + 1$          // benign race
29                  BAG-INSERT($V_{d+1}, v$)
30      return
31  $N'$ = PENNANT-SPLIT($N$)
32  spawn PROCESS-PENNANT($N', V_{d+1}, d$)
33  PROCESS-PENNANT($N, V_{d+1}, d$)
34  sync
```

Figure 3-3: The PBFS algorithm operating on a graph G with source vertex $v_0 \in V(G)$. PBFS uses the parallel subroutine PROCESS-LAYER to process each layer, which is described in detail in Section 3.3. PBFS contains a benign race in line 28.

```
28.1  if TRY-LOCK($v$)
28.2      if $v.dist == \infty$
28.3          $v.dist = d + 1$
28.4          BAG-INSERT($V_{d+1}, v$)
28.5          RELEASE-LOCK($v$)
```

Figure 3-4: Modification to the PBFS algorithm to resolve the benign race.

hence no inconsistency arises from updating the location at the same time. Both processors go on to insert v into bag V_{d+1} in line 29, which could induce another race. Putting that issue aside for the moment, notice that inserting multiple copies of v into V_{d+1} does not affect correctness, only performance for the extra work it will take when processing layer $d + 1$, because v will be encountered multiple times. As Section 3.4 discusses, the amount of extra work is small, because the race is rarely actualized.

Second, a race in line 29 occurs due to parallel insertions of vertices into V_{d+1}. We employ the reducer functionality to avoid the race by making V_{d+1} a bag reducer (via the **reducer** keyword on line 18), where BAG-UNION is the associative operation required by the reducer mechanism. The identity for BAG-UNION — an empty bag — is created by BAG-CREATE. In the common case, line 29 simply inserts v into the local view, which,

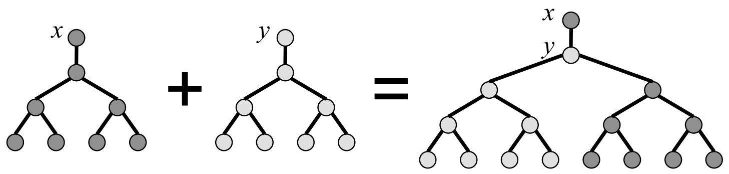


Figure 3-5: Illustration of the PENNANT-UNION operation, which combines two pennants, each of size 2^k, in constant time to form a pennant of size 2^{k+1}.

as Section 3.3 shall show, is generally as efficient as pushing v onto a FIFO, as is done by SERIAL-BFS. To process layer V_{d+1} in the next iteration, the **revert** keyword on line 19 converts V_{d+1} from a reducer into an ordinary bag that can PROCESS-LAYER can read in parallel.

Unfortunately, we are not able to analyze PBFS due to unstructured nondeterminism created by the benign race, but we can analyze a version where the race is resolved using a mutual-exclusion lock. The locking version involves replacing lines 28 and 29 with the code in Figure 3-4. In the code, the call TRY-LOCK(v) in line 28.1 attempts to acquire a lock on the vertex v. If it is successful, we proceed to execute lines 28.2–28.5. Otherwise, we can abandon the attempt, because we know that some other processor has succeeded, and thus $v.\mathit{dist}$ will be set to $d+1$ regardless. There is therefore no contention on v's lock, because no processor ever waits for another, and processing an edge (u, v) always takes constant time. The apparently redundant lines 27 and 28.2 avoid the overhead of lock acquisition when $v.\mathit{dist}$ has already been set.

3.3 The bag data structure

This section describes the bag data structure for implementing a dynamic unordered set. We describe an auxiliary data structure called a "pennant." We show how bags can be implemented using pennants, and we provide algorithms for BAG-CREATE, BAG-INSERT, and BAG-UNION. We describe how to extract the elements in a bag in parallel. We discuss some optimizations of this structure that PBFS employs.

Pennants

A ***pennant*** is a tree of 2^k nodes, where k is a nonnegative integer. Each node x in this tree contains two pointers $x.\mathit{left}$ and $x.\mathit{right}$ to its children. The root of the tree has only a left child, which is a complete binary tree of the remaining elements.

Two pennants x and y of size 2^k can be combined to form a pennant of size 2^{k+1} in $\Theta(1)$ time using the following PENNANT-UNION function, which is illustrated in Figure 3-5.

```
PENNANT-UNION($x, y$)
35  $y.right = x.left$
36  $x.left = y$
37  return $x$
```

The function PENNANT-SPLIT performs the inverse operation of PENNANT-UNION in $\Theta(1)$ time. Given a pennant x containing at least 2 elements, PENNANT-SPLIT modifies x and returns a new pennant y such that each of x and y contain half of the elements originally in x.

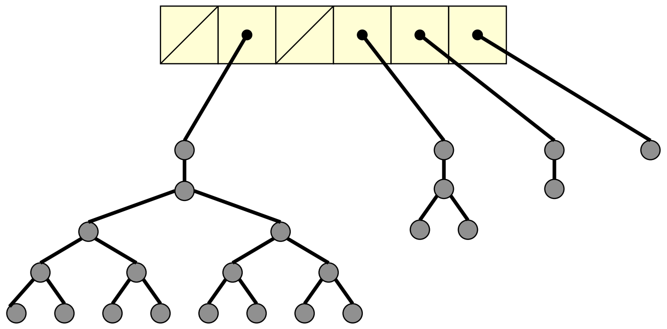


Figure 3-6: A bag with $23 = 010111_2$ elements.

```
PENNANT-SPLIT($x$)
$y = x.left$
$x.left = y.right$
$y.right$ = NULL
return $y$
```

Bags

A ***bag*** is a collection of pennants, no two of which have the same size. PBFS represents a bag S using a fixed-size array $S[0\,..\,r]$, called the ***backbone***, where 2^{r+1} exceeds the maximum number of elements ever stored in a bag. Each entry $S[k]$ in the backbone contains either a null pointer or a pointer to a pennant of size 2^k. Figure 3-6 illustrates a bag containing 23 elements, for example.

The BAG-CREATE function allocates space for a fixed-size backbone of null pointers, which takes $\Theta(r)$ time. This bound can be improved to $\Theta(1)$ by keeping track of the largest nonempty index in the backbone.

The BAG-INSERT function employs an algorithm similar to that of incrementing a binary counter. To implement BAG-INSERT, the given element is first packaged as a pennant x of size 1. We then insert x into bag S using the following method.

```
BAG-INSERT($S, x$)
$k = 0$
while $S[k] \neq$ NULL
    $x$ = PENNANT-UNION($S[k], x$)
    $S[k{+}{+}]$ = NULL
$S[k] = x$
```

The analysis of BAG-INSERT mirrors the analysis for incrementing a binary counter [100, Ch. 17]. Since every PENNANT-UNION operation takes constant time, BAG-INSERT takes $\Theta(1)$ amortized time and $\Theta(\lg n)$ worst-case time to insert into a bag containing n elements.

The BAG-UNION function uses an algorithm similar to ripple-carry addition of two binary counters. To implement BAG-UNION, we first examine the process of unioning three pennants into two pennants, which operates like a full adder. Three pennants x, y, and z, where each either has size 2^k or is empty, can be merged to produce a pair of pennants (s, c), where s has size 2^k or is empty, and c has size 2^{k+1} or is empty. The following table details the function FA(x, y, z) in which (s, c) is computed from (x, y, z), where 0 means that the designated pennant is empty, and 1 means that it has size 2^k:

x	y	z	s	c
0	0	0	NULL	NULL
1	0	0	x	NULL
0	1	0	y	NULL
0	0	1	z	NULL
1	1	0	NULL	PENNANT-UNION(x, y)
1	0	1	NULL	PENNANT-UNION(x, z)
0	1	1	NULL	PENNANT-UNION(y, z)
1	1	1	x	PENNANT-UNION(y, z)

With this full-adder function in hand, BAG-UNION can be implemented as follows:

```
BAG-UNION($S_1, S_2$)
47  $y$ = NULL   // The "carry" bit.
48  for $k = 0$ to $r$
49      $(S_1[k], y) = $ FA$(S_1[k], S_2[k], y)$
```

Because every PENNANT-UNION operation takes constant time, computing the value of FA(x, y, z) also takes constant time. To compute all entries in the backbone of the resulting bag takes $\Theta(r)$ time. This algorithm can be improved to $\Theta(\lg n)$, where n is the total number of elements in the two bags, by maintaining the largest nonempty index of the backbone of each bag and unioning the bag with the smaller such index into the one with the larger.

Given this design for the bag data structure, let us now return to the pseudocode for PROCESS-LAYER and PROCESS-PENNANT in Figure 3-3. To process the elements of V_d, PROCESS-LAYER calls PROCESS-PENNANT on each non-null pennant N in V_d (on lines 21–23) in parallel. PROCESS-PENNANT processes pennant N using parallel divide-and-conquer over its elements. For the recursive case, line 31 splits N, removing half of its elements and placing them in N'. Lines 32 and 33 process the two halves recursively in parallel.

This recursive decomposition continues until N has fewer than GRAINSIZE elements, which line 24 checks. Each vertex u in N is extracted in line 25, and as described in Section 3.2, line 26 examines each of its edges (u, v) in parallel. If v has not yet been visited — line 27 discovers that $v.dist$ is infinite — then line 28 sets $v.dist = d + 1$ and line 29 inserts v into bag V_{d+1}.

Optimization

To improve the constant in the performance of BAG-INSERT, we made some simple but important modifications to pennants and bags, which do not affect the asymptotic behavior of the algorithm. First, in addition to its two pointers, every pennant node in the bag stores a constant-size array of GRAINSIZE elements, all of which are guaranteed to be valid, rather than just a single element. Our PBFS software uses the value GRAINSIZE = 128. Second, in addition to the backbone, the bag itself maintains an additional pennant node of size GRAINSIZE called the ***hopper***, which it fills gradually. These modifications impact the bag operations and PBFS algorithm as follows.

First, BAG-CREATE allocates additional space for the hopper. This overhead is small as the allocation is done only once per bag.

Second, BAG-INSERT targets the hopper when inserting an element, rather than the pennants in the backbone of the bag. If the hopper is full, then it inserts the full hopper into the backbone of the bag and allocates a new hopper into which it inserts the element. This optimization does not change the asymptotic running-time analysis of BAG-INSERT, but the code runs much faster. In the common case, BAG-INSERT simply inserts the element into the hopper with code nearly identical to that for inserting an element into a FIFO. Only once in every GRAINSIZE insertions does a BAG-INSERT trigger the insertion of the now full hopper into the backbone of the bag.

Third, when unioning two bags S_1 and S_2, BAG-UNION also combines their hoppers. BAG-UNION first determines which bag has the less full hopper. For this description, let us assume that S_1 has the less full hopper, since the alternative case is symmetric. BAG-UNION then copies the elements of S_1's hopper into S_2's hopper until either S_2's hopper is full or S_1's hopper is empty. If it empties S_1's hopper, then BAG-UNION proceeds to merge the two bags as usual and uses S_2's hopper as the hopper for the resulting bag. If it fills S_2's hopper, however, line 47 of BAG-UNION sets y to S_2's hopper and sets S_1's hopper, now containing fewer elements, to be the hopper for the resulting bag before proceeding as usual.

Finally, the **for** loop on lines 21–23 in PROCESS-LAYER additionally calls PROCESS-PENNANT on a unit-sized pennant containing the hopper of bag V_d.

3.4 Experimental results

We implemented optimized versions of both the PBFS algorithm in Cilk++ and SERIAL-BFS in C++. This section compares their performance on a suite of benchmark graphs. Figure 3-7 summarizes the results.

Implementation and testing

Our implementation of PBFS differs from the abstract algorithm in some notable ways. First, our implementation of PBFS does not use locks to resolve the benign races described in Section 3.2. Second, our implementation assumes that all vertices have bounded out-degree, and indeed most of the vertices in our benchmark graphs have relatively small degree. Finally, our implementation of PBFS sets GRAINSIZE $= 128$, which seems to perform well in practice. The SERIAL-BFS implementation uses an array, sized to the number of vertices in the input graph, and two pointers to implement the FIFO queue in the simplest way possible.

These implementations were tested on the eight benchmark graphs described in Figure 3-7. Kkt_power, Cage14, Cage15, Freescale1, Wikipedia (as of February 6, 2007), and Nlpkkt160 are all from the University of Florida sparse-matrix collection [107]. Grid3D200 is a 7-point finite difference mesh generated using the Matlab Mesh Partitioning and Graph Separator Toolbox [159]. The RMat23 matrix [251], which models scale-free graphs, was generated by using repeated Kronecker products [28]. Parameters $A = 0.7$, $B = C = D = 0.1$ for RMat23 were chosen in order to generate skewed matrices. We stored these graphs in a compressed-sparse-rows format [374] in main memory for our empirical tests.

We ran our experiments on the machine described in Figure 3-8. All code was compiled using `-O3` optimization.

Results

Figure 3-7 presents the performance of PBFS on eight different benchmark graphs. (The parallelism was computed using the Cilkview [180] tool and does not take into account effects from reducers.) As can be seen in Figure 3-7, PBFS performs well on these benchmark

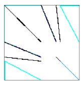
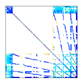
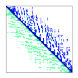
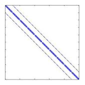
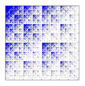
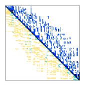
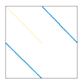
Name *Description*	*Spy Plot*	$\|V\|$ $\|E\|$ D	*Work* *Span* *Parallelism*	SERIAL-BFS T_1 PBFS T_1 PBFS T_1/T_8
Kkt_power Optimal power flow, nonlinear opt.		2.05 M 12.76 M 31	241 M 2.3 M 103.85	0.504 0.359 5.983
Freescale1 Circuit simulation		3.43 M 17.1 M 128	349 M 2.3 M 152.72	0.285 0.327 5.190
Cage14 DNA electrophoresis		1.51 M 27.1 M 43	390 M 1.6 M 245.70	0.262 0.283 5.340
Wikipedia Links between Wikipedia pages		2.4 M 41.9 M 460	606 M 3.4 M 178.73	0.914 0.721 6.381
Grid3D200 3D 7-point finite-diff mesh		8 M 55.8 M 598	1 009 M 12.7 M 79.27	1.544 1.094 4.862
RMat23 Scale-free graph model		2.3 M 77.9 M 8	1 050 M 11.3 M 93.22	1.100 0.936 6.500
Cage15 DNA electrophoresis		5.15 M 99.2 M 50	1 410 M 2.1 M 674.65	1.065 1.142 5.263
Nlpkkt160 Nonlinear optimization		8.35 M 225.4 M 163	3 060 M 9.2 M 331.45	1.269 1.448 5.983

Figure 3-7: Performance results for breadth-first search. The vertex and edge counts listed correspond to the number of vertices and edges evaluated by SERIAL-BFS, while D denotes the length of the longest shortest path discovered by SERIAL-BFS. The work and span are measured in instructions. All running times are measured in seconds.

graphs. For five of the eight benchmark graphs, PBFS is as fast or faster than SERIAL-BFS. Moreover, on the remaining three benchmarks, PBFS is at most 15 % slower than SERIAL-BFS.

This performance advantage PBFS holds over SERIAL-BFS on several of the benchmark graphs might be due to how PBFS uses memory. Whereas SERIAL-BFS performs a single linear scan through an array as it processes its queue, PBFS is constantly allocating and deallocating fixed-size chunks of memory of size GRAINSIZE for the bag. Because these chunks do not change in size from allocation to allocation, the memory manager incurs little work to perform these allocations. Perhaps more importantly, PBFS can frequently reuse previously allocated chunks, making it more cache-friendly. This improvement due to memory reuse is also apparent in some serial BFS implementations that use two queues instead of one.

Although PBFS generally performs well on these benchmarks, we explored why it was only attaining a speedup of 5 or 6 on 8 processor cores. Inadequate parallelism is not the answer, as most of the benchmarks have over 100 parallelism. Our studies indicate that the multicore processor's memory system may be hurting performance in two ways.

CPU	Intel Core i7
Clock	2.53 GHz
Hyperthreading	Disabled
Cores per processor chip	4
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	256 KiB
L3 cache/socket	8 MiB
DRAM	12 GiB
Compiler	Cilk++ compiler

Figure 3-8: Technical specifications of the machine used for benchmarking.

First, the memory bandwidth of the system seems to limit performance for several of these graphs. For Wikipedia and Cage14, when we run 8 independent instances of PBFS serially on the 8 processor cores of our machine simultaneously, the total runtime is at least 20 % worse than the expected $8T_1$. This experiment suggests that the system's available memory bandwidth limits the performance of the parallel execution of PBFS.

Second, for several of these graphs, contention from true and false sharing on the distance array appears to constrain speedup. Placing each location in the distance array on a different cache line tends to increase the speedups somewhat, although it slows down overall performance due to the loss of spatial locality. We attempted to modify PBFS to mitigate contention by randomly permuting or rotating each adjacency list. Although these approaches improve the observed speedups, they slow down overall performance due to loss of locality. Despite its somewhat lower relative speedup numbers, therefore, the unadulterated PBFS seems to yield the best overall performance.

PBFS obtains good performance despite the benign race which induces redundant work. On none of these benchmarks does PBFS examine more than 1 % of the vertices and edges redundantly. Using a mutex lock on each vertex to resolve the benign race costs a substantial overhead in performance, typically slowing down PBFS by more than a factor of 2.

Yuxiong He [179], formerly of Cilk Arts and Intel Corporation, used PBFS to parallelize the Murphi model-checking tool [118]. Murphi is a popular tool for verifying finite-state machines and is widely used in cache-coherence algorithms and protocol design, link-level protocol design, executable memory-model analysis, and analysis of cryptographic and security-related protocols. As can be seen in Figure 3-9, a parallel Murphi using PBFS scales well, even outperforming a version based on parallel depth-first search and attaining the relatively large speedup factor of 15.5 on 16 cores.

3.5 Modeling reducers

This section extends the dag model of dynamic multithreading presented in Chapter 2 to incorporate reducer hyperobjects. Although the parallel speedup of deterministic dynamic multithreaded programs can be bounded theoretically using work-span analysis, as presented in Chapter 2, obtaining bounds on the speedup of a program such as PBFS can be more challenging, because of its use of reducer hyperobjects. In particular, programs that use reducers are nondeterministic because the runtime system creates and reduces views based on scheduling happenstance. For such nondeterministic programs, the work of a P-processor execution might not readily be related to the serial running time. We define the notion of a "user dag" for a computation, which represents the strands that are visible to the

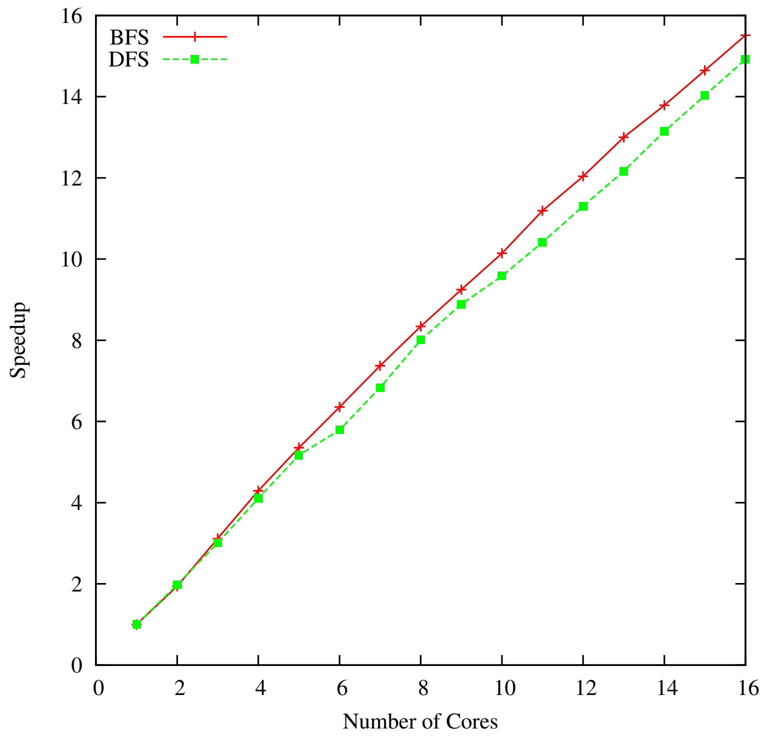


Figure 3-9: Multicore Murphi application speedup on a 16-core AMD processor [179]. Even though the DFS implementation uses a parallel depth-first search for which Cilk++ is particularly well suited, the BFS implementation, which uses the PBFS library, outperforms it.

programmer. We also define the notion of a "performance dag," which includes the strands that the runtime system implicitly invokes.

To analyze PBFS, we think of the bag data structure, like other reducers, as an algebraic monoid. In the case of bags, the REDUCE function is BAG-UNION, the CREATE-IDENTITY function is BAG-CREATE, and the UPDATE function is BAG-INSERT. Although the REDUCE operator for bags is not actually associative, bags admit an idea of "logical" associativity, which is sufficient in practice. If we have three bags S_1, S_2, and S_3, we do not care whether the bag data structures for $(S_1 \cup S_2) \cup S_3$ and $S_1 \cup (S_2 \cup S_3)$ are identical, only that they contain the same elements.

To specify the nondeterministic behavior encapsulated by reducers precisely, consider a computation A from running a dynamic multithreaded program, and let $V(A)$ be the set of executed strands. We assume that the implicitly invoked functions for a reducer — REDUCE and CREATE-IDENTITY — execute only serial code. We model each execution of one of these functions as a single strand containing the instructions of the function. If an UPDATE causes the runtime system to invoke CREATE-IDENTITY implicitly, the serial code arising from UPDATE is broken into two strands sandwiching the point where CREATE-IDENTITY is invoked.

We partition $V(A)$ into three classes of strands.

- V_ι: ***Init strands*** arise from the execution of CREATE-IDENTITY when invoked implicitly by the runtime system, which occur when the user program attempts to update a reducer, but a local view has not yet been created.
- V_ρ: ***Reduce strands*** arise from the execution of REDUCE, which occur implicitly when the runtime system combines views.
- V_ν: ***User strands*** arise from the execution of code explicitly invoked by the program-

mer, including calls to UPDATE.

We call $V_\iota \cup V_\rho$ the set of ***runtime strands***.

Because the programmer sees runtime strands as invoked "invisibly" by the runtime system, his or her understanding of the program generally relies only on the user strands. We capture the control dependencies among the user strands by defining the ***user dag*** $A_\nu = (V_\nu, E_\nu)$ for a computation A in the same manner as we defined an ordinary multithreaded dag. For example, a spawn strand e_1 has out-degree 2 in A_ν with an edge (v_1, v_2) going to the first strand v_2 of the spawned child and the other edge (v_2, v_3) going to the continuation v_3; if v_1 is the final strand of a spawned subroutine and v_2 is the sync strand with which v_1 syncs, then we have $(v_1, v_2) \in E_\nu$; etc.

To track the views of a reducer h in the user dag, let $h(v)$ denote the view of h that a strand $v \in V_\nu$ can access. The runtime system maintains the following invariants.

Invariant 1 *If $u \in V_\nu$ has out-degree 1 and $(u, v) \in E_\nu$, then $h(v) = h(u)$.*

Invariant 2 *Suppose that $u \in V_\nu$ is a spawn strand with outgoing edges $(u, v), (u, w) \in E_\nu$, where $v \in V_\nu$ is the first strand of the spawned subroutine and $w \in V_\nu$ is the continuation in the parent. Then, we have $h(v) = h(u)$ and*

$$h(w) = \begin{cases} h(u) & \text{if } u \text{ was not stolen} \\ \text{new view} & \text{otherwise.} \end{cases}$$

Invariant 3 *If $v \in V_\nu$ is a sync strand, then $h(v) = h(u)$, where u is the first strand of v's function.*

When a new view $h(w)$ is created, as is inferred by Invariant 2, we say that the old view $h(u)$ ***dominates*** $h(w)$, which we denote by $h(u) > h(w)$. For a set H of views, we say that two views $h_1, h_2 \in H$, where $h_1 > h_2$, are ***adjacent*** if there does not exist $h_3 \in H$ such that $h_1 > h_3 > h_2$.

A useful property of sync strands is that the views of strands entering a sync strand $v \in V_\nu$ are totally ordered by the dominates relation. That is, if k strands each have an edge in E_ν to the same sync strand $v \in V_\nu$, then the strands can be numbered $u_1, u_2, \ldots, u_k \in V_\nu$ such that $h(u_1) \geq h(u_2) \geq \cdots \geq h(u_k)$. Moreover, $h(u_1) = h(v) = h(u)$, where u is the first strand of v's function. These properties can be proved inductively, noting that the views of the first and last strands of a function must be identical, because a function implicitly syncs before it returns. The runtime system always reduces adjacent pairs of views in this ordering, destroying the dominated view in the pair.

If a computation A does not involve any runtime strands, then the "delay-sequence" argument in [58] can be applied to A_ν to bound the P-processor execution time: $T_P(A) \leq \text{Work}(A_\nu)/P + O(\text{Span}(A_\nu))$. Our goal is to apply this same analytical technique to computations containing runtime strands. To do so, we augment the A_ν with the runtime strands to produce a ***performance dag*** $A_\pi = (V_\pi, E_\pi)$ for the computation A, where

- $V_\pi = V(A) = V_\nu \cup V_\iota \cup V_\rho$, and
- $E_\pi = E_\nu \cup E_\iota \cup E_\rho$,

where the edge sets E_ι and E_ρ are constructed as follows.

The edges in E_ι are created in pairs. For each init strand $v \in V_\iota$, we include (u, v) and (v, w) in E_ι, where $u, w \in V_\nu$ are the two strands comprising the instructions of the UPDATE whose execution caused the invocation of the CREATE-IDENTITY corresponding to v.

The edges in E_ρ are created in groups corresponding to the set of REDUCE functions that must execute before a given sync. Suppose that $v \in V_\nu$ is a sync strand, that k strands $u_1, u_2, \ldots, u_k \in A_\nu$ join at v, and that $k' < k$ reduce strands $r_1, r_2, \ldots, r_{k'} \in A_\rho$ execute before the sync. Consider the set $U = \{u_1, u_2, \ldots, u_k\}$, and let $h(U) = \{h(u_1), h(u_2), \ldots, h(u_k)\}$ be the set of $k'+1$ views that must be reduced. The edges in E_ρ connect the strands in U, the strands $r_1, r_2, \ldots, r_{k'}$, and the strand v together into a ***reduce tree*** as follows:

```
50  while $|h(U)| \geq 2$
51      Let $r \in \{r_1, r_2, \ldots, r_{k'}\}$ be the reduce strand that reduces a "minimal"
        pair $h_j, h_{j+1} \in h(U)$ of adjacent strands, meaning that if a distinct $r' \in$
        $\{r_1, r_2, \ldots, r_{k'}\}$ reduces adjacent strands $h_i, h_{i+1} \in h(U)$, we have $h_i > h_j$
52      Let $U_r = \{u \in U : h(u) = h_j$ or $h(u) = h_{j+1}\}$
53      Include in $E_\rho$ the edges in the set $\{(u, r) : u \in U_r\}$
54      $U = U - U_r \cup \{r\}$
55  Include in $E_\rho$ the edges in the set $\{(r, v) : r \in U\}$
```

Since the reduce trees and init strands only add more dependencies between strands in the user A_ν that are already in series, the performance dag A_π is indeed a dag.

3.6 Analysis of programs with nonconstant-time reducers

This section provides a framework for analyzing dynamic multithreaded programs that might use a ***nonconstant-time reducer*** — a reducer whose REDUCE function takes $\Omega(1)$ time to execute. Much like Blumofe and Leiserson's analysis of a randomized work-stealing scheduler [58], we use an accounting argument to bound the expected running time of a computation A with performance dag A_π. We describe how the Cilk runtime system maintains reducer views in detail. We bound the work and span of A_π in terms of the work and span of the user dag A_ν and the worst-case running time τ of any REDUCE or CREATE-IDENTITY operation. Combining these bounds, we prove that the expected running time of A is $T_P(A) \leq \text{Work}(A_\nu)/P + O(\tau^2 \cdot \text{Span}(A_\nu))$.

Throughout this section, we shall refer to dynamic multithreaded programs and computations simply as multithreaded programs and computations.

Scheduling with reducers

Let us first formally define a schedule. Executing a computation A on a P-processor computer using a work-stealing scheduler produces a ***schedule*** of A, which is a mapping $C : V(A) \to \{0, 1, \ldots, P-1\} \times \mathbb{N}$ where, for each strand $u \in V(A)$, if $C(u) = (w_u, t_u)$ then worker w_u begins executing u at time step t_u. A schedule must obey all control dependencies in the dag of A, meaning that, if there exists a path from a strand u to a strand v in A, then we have $C(u) = (w_u, t_u)$ and $C(v) = (w_v, t_v)$ such that $t_u < t_v$. If a strand u depends on every strand in a set $\{v_1, v_2, \ldots, v_k\} \subseteq V(A)$, we say that the last strand in $\{v_1, v_2, \ldots, v_k\}$ to finish executing ***enables*** u.

Our analysis of the performance of a multithreaded program that uses a reducer relies on the runtime system joining strands quickly even when it must invoke nonconstant-time REDUCE operations. Providing a guarantee on the time to join strands requires that we examine the specifics of how the runtime system handles reducers.

Let us review how the runtime system handles spawns and steals, as described by Frigo *et al.* [144]. Every time a Cilk function is stolen, the runtime system creates a new ***frame***.[2] Although frames are created and destroyed dynamically during a program execution, the ones that exist always form a rooted ***spawn tree***. Each frame F provides storage for temporary values and local variables, as well as metadata for the function, including the following:

- A pointer $F.\mathit{lp}$ to either F's left sibling or, if F is the first child, to F's parent;
- A pointer $F.c$ to F's first child;
- A pointer $F.r$ to F's right sibling.

These pointers form a left-child right-sibling representation of the part of the spawn tree that is distributed among processors, which is known as the ***steal tree***.

To handle reducers, each worker in the runtime system uses a hash table, called a ***hypermap***, to map reducers to its local views. To allow for lock-free access to the hypermap of a frame F while siblings and children of the frame are terminating, F stores three hypermaps, denoted $F.\mathit{hu}$, $F.\mathit{hr}$, and $F.\mathit{hc}$. The $F.\mathit{hu}$ hypermap is used to look up reducers for the user's program, while the $F.\mathit{hr}$ and $F.\mathit{hc}$ hypermaps store the accumulated values of F's terminated right siblings and terminated children, respectively.

A frame's hypermaps are empty when it is initially created. If a worker using a frame F executes an UPDATE operation on a reducer h, then the worker tries to get h's current view from the $F.\mathit{hu}$ hypermap. If h's view is empty, then the worker performs a CREATE-IDENTITY operation to create an identity view of h in $F.\mathit{hu}$.

When a worker returns from a spawn, first it must perform up to two REDUCE operations to reduce its hypermaps into its neighboring frames, and then it must ***eliminate*** its current frame. To perform these REDUCE operations and elimination without races, the worker grabs locks on its neighboring frames. The algorithm by Frigo *et al.* [144] uses an intricate protocol to avoid long waits on locks, but the analysis of its performance assumes that each REDUCE takes only constant time.

To support nonconstant-time REDUCE functions, we modify the locking protocol. To eliminate a frame F, the worker first reduces $F.\mathit{hu} \otimes= F.\mathit{hr}$. Second, the worker reduces $F.\mathit{lp}.\mathit{hc} \otimes= F.\mathit{hu}$ or $F.\mathit{lp}.\mathit{hr} \otimes= F.\mathit{hu}$, depending on whether or not F is a first child.

Workers eliminating $F.\mathit{lp}$ and $F.r$ might race with the elimination of F. To resolve these races, Frigo *et al.* describe how to acquire an abstract lock between F and these neighbors, where an abstract lock is a pair of locks that correspond to an edge in the steal tree. We shall use these abstract locks to eliminate a frame F according to the locking protocol shown in Figure 3-10. As Lemma 9 will show, this locking protocol ensures that locks are only held for $\Theta(1)$ time, regardless of the time required to execute a REDUCE operation.

Analyzing a performance dag

We next bound the time a work-stealing scheduler takes to execute a multithreaded computation A that might use a nonconstant-time reducer in terms of its performance dag A_π. The proof follows that of Blumofe and Leiserson [58] and that of Frigo *et al.* [144], with some salient differences to handle init and reduce strands. For simplicity, let us assume that computation A makes use of at most a single reducer. The following proofs can be extended to handle many reducers within a computation.

[2]When we refer to frames in this chapter, we specifically mean the "full" frames described by Frigo *et al.* [144].

```
while TRUE
    Acquire the abstract locks for edges $(F, F.lp)$ and $(F, F.r)$ in an order
    chosen uniformly at random
    if $F$ is a first child
        $L = F.lp.hc$
    else $L = F.lp.hr$
    $R = F.hr$
    if $L == \emptyset$ and $R == \emptyset$
        if $F$ is a first child
            $F.lp.hc = F.hu$
        else $F.lp.hr = F.hu$
        Eliminate $F$
        Release the abstract locks
        break
    $R' = R$; $L' = L$
    $R = \emptyset$; $L = \emptyset$
    Release the abstract locks
    for each reducer $h \in R'$
        if $h \in F.hu$
            $F.hu(h) \otimes= R'(h)$
        else $F.hu(h) = R'(h)$
    for each reducer $h \in L'$
        if $h \in F.hu$
            $F.hu(h) = L'(h) \otimes F.hu(h)$
        else $F.hu(h) = L'(h)$
```

Figure 3-10: A modified locking protocol for managing reducers, which holds locks for $\Theta(1)$ time.

We use an accounting argument to analyze the running time of a work-stealing scheduler executing a multithreaded computation. To model the process of running a multithreaded computation, let us represent a unit of computation performed by a processor during a time step with a dollar. At each time step, each processor spends its dollar by placing it into some bucket, depending on the type of task that processor performs at the step. If the execution takes time T_P, then at the end the total number of dollars in all the buckets is PT_P. Consequently, if we sum up all of the dollars in all of the buckets and divide by P, we obtain the running time.

We consider four different types of tasks a processor can perform at a step. If a processor executes an instruction at the step, then it places its dollar into the WORK bucket. If a processor initiates a steal attempt at the step, then it places its dollar into the STEAL bucket. If a processor waits for its steal request to be satisfied at the step, then it places its dollar into the WAIT-STEAL bucket. Finally, if a processor waits to acquire a lock on a data structure in the runtime system at the step, then it places its dollar into the WAIT-LOCK bucket.

We first bound the total number of dollars in the WORK bucket.

Lemma 4 *The execution of a dynamic multithreaded computation A with work* $\text{Work}(A_\pi)$ *by a randomized work-stealing scheduler on a computer with P processors, terminates with exactly* $\text{Work}(A_\pi)$ *dollars in the* WORK *bucket.*

PROOF. Because there are exactly $\text{Work}(A_\pi)$ instructions in the computation, the execution ends with exactly $\text{Work}(A_\pi)$ dollars in the WORK bucket. □

Next, we bound the total dollars in the STEAL bucket using a delay-sequence argument, as presented by Ranade [326]. The main idea of the argument is to first construct a "delay sequence" with the performance dag, and to then argue that any delay sequence has a low probability of occurring. This proof resembles that of Blumofe and Leiserson [58], but with care given to handle init and reduce strands.

To begin this argument, we identify a "delaying path," which is a path of "critical strands," through an augmented performance dag A_ϕ. The augmented performance dag represents both control dependencies and "deque dependencies" on $V(A)$. To construct a delay sequence, we divide the set of steal attempts that occur during A's execution into rounds, and we associate each round with some strand on the delaying path. Next, we bound the probability that a strand remains critical across many steal attempts. We conclude that any delay sequence is unlikely to occur and the execution of A is therefore unlikely to suffer a large number of steal attempts.

Let us assume without loss of generality that every strand represents a single instruction that executes in a single time step. When we refer to an init or reduce strand, we mean the sequence of single-instruction strands that perform a particular CREATE-IDENTITY or REDUCE operation.

A ***round*** of steal attempts is a set of at least $3P$ but fewer than $4P$ consecutive steal attempts such that if a steal attempt that is initiated at time step t occurs in a particular round, then all other steal attempts initiated at time step t occur in the same round. We can partition all of the steal attempts that occur during an execution as follows. The first round contains all steal attempts initiated at time steps $1, 2, \ldots, t_1$, where t_1 is the first time step such that at least $3P$ steal attempts were initiated at or before t_1. For $i > 1$ we say that the ith round of steals begins at time step $t_{i-1} + 1$ and ends at time step t_i, where at least $3P$ consecutive steal attempts were initiated between time step $t_{i-1} + 1$ and time step t_i. Each round necessarily contains at least $3P$ consecutive steal attempts, and because fewer than P steal attempts can be initiated at a single time step, each round contains fewer than $4P$ steal attempts, and each round takes at least 4 steps.

The criticality of a strand $v \in V(A)$ is defined in terms of control dependencies and "deque dependencies," which are defined on the successors of spawn strands. A ***deque dependency*** exists from a strand u to a strand v if v is the first child strand after a spawn strand w and u is the continuation strand after w. We augment a given performance dag A_π to form a new dag A_ϕ that reflects both the control and deque dependencies between strands. Formally, the augmented performance dag A_ϕ contains all of the vertices and edges of the performance dag A_π plus an edge (u, v) for every pair of strands $u, v \in V(A)$ for which a deque dependency exists from u to v. Because we assume that no continuation strand is also a sync strand, the augmented scheduling dag is indeed a dag, and we have $\text{Span}(A_\phi) \leq 2 \cdot \text{Span}(A_\pi)$. We assume for simplicity that neither the first spawned child of a spawn strand w nor the continuation after w is a reduce strand.

A strand $u \in V(A)$ is ***critical*** at time step t if it is unexecuted at time step t and all immediate predecessor strands $v \in \{v_1, v_2, \ldots, v_k\}$ of u in A_ϕ have been executed. If a strand u is critical, then u must be enabled, because all strands with control dependencies into u have been executed.

We use the augmented performance dag A_ϕ to create a delay sequence. A ***delay sequence*** of a computation A with schedule C is a triple (p, R, Π) consisting of the following:

- A ***delaying path*** $p = \langle u_1, u_2, \ldots, u_L \rangle$ — a maximal path through A_ϕ, in which strand u_1 is the first strand in A_ϕ, strand u_L is the last strand in A_ϕ, and for each $i = 1, 2, \ldots, L-1$, strand u_{i+1} follows strand u_i if $(u_i, u_{i+1}) \in E(A_\phi)$.
- A positive number R of steal-attempt rounds.
- A partition $\Pi = (\pi_1, \pi_1', \pi_2, \pi_2', \ldots, \pi_L, \pi_L')$ of R (that is $R = \sum_{i=1}^{L}(\pi_i + \pi_i')$), such that

$\pi'_i \in \{0, 1\}$ for $i = 1, 2, \ldots, L$.

For each $i = 1, 2, \ldots, L$, we define the ith ***group*** of steal-attempt rounds to be the π_i consecutive rounds that begin after the r_ith round, where $r_i = \sum_{j=1}^{i-1} \pi_j \cup \pi'_j$. Because Π is a partition of R and $\pi'_i \in \{0, 1\}$ for $i = 1, 2, \ldots, L$, we have

$$\sum_{i=1}^{L} \pi_i \geq R - L \ , \tag{3.1}$$

We say that a given round of steal attempts ***occurs*** while strand v is critical if all of the steal attempts that comprise the round are initiated at time steps when v is critical. In other words, v must be critical throughout the entire round. A delay sequence ***occurs*** during the execution of A if, for each $i = 1, 2, \ldots, L$, strand u_i is critical throughout all π_i rounds in the ith group.

The following lemma shows that, if at least R rounds take place during an execution, then some delay sequence (p, R, Π) must occur. In particular, in any execution in which at least R rounds occur, we can identify a delaying path $p = \langle u_1, u_2, \ldots, u_L \rangle$ in A_ϕ and a partition $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \ldots, \pi_L, \pi'_L)$ of R such that, for each $i = 1, 2, \ldots, L$, all of the π_i rounds in ith group occur while u_i is critical. Each π'_i counts at most one round, which is the round that began when u_i was critical and ended after u_i began executing. Such a round cannot be part of any group because no instruction is critical throughout.

Lemma 5 *Consider an execution of a dynamic multithreaded computation A by a randomized work-stealing scheduler on an ideal parallel computer with P processors. If at least $4PR$ steal attempts occur during A's execution, then some delay sequence (p, R, Π) must occur.*

Proof. The proof follows from the techniques of Blumofe and Leiserson [58, Lemma 9] for constructing a delaying path, using A_ϕ as the dag. $\square$

We now argue that a critical instruction is unlikely to remain critical across a modest number of rounds. In particular, we argue that the analysis of Blumofe and Leiserson [58] applies to performance dags containing runtime strands.

Lemma 6 *Consider the execution of a dynamic multithreaded computation A by a randomized work-stealing scheduler on an ideal parallel computer with $P \geq 2$ processors. For any strand u and any number $r \geq 2$ of steal-attempt rounds, the probability that any particular set of r rounds occur while the strand u is critical is at most the probability that only 0 or 1 of the steal attempts initiated in the first $r - 1$ of these rounds chose u's worker, which is at most e^{-2r+3}.*

Proof. Let C be the schedule generated by the execution of A. If u is a user strand, then this claim follows from the argument by Blumofe and Leiserson [58, Lemma 11]. Suppose therefore that u is a runtime strand, and let $C(u) = (w_u, t_u)$. In this case, because runtime strands are executed opportunistically by the scheduler, u was never placed on w_u's deque. Instead, w_u begins executing u at the time step after u is enabled. Because at most $P - 1$ steal attempts can occur during a single time step and a round of steal attempts consists of at least $3P$ steal attempts, less than 1 round of steal attempts can occur while u is critical. Therefore, u is not critical throughout any rounds of steal attempts, and thus the probability bound of Blumofe and Leiserson [58, Lemma 11] holds. $\square$

The following lemma now completes the delay-sequence argument and bounds the total dollars in the Steal bucket. The proof of this lemma largely follows the analysis of Blumofe and Leiserson [58, Lemma 12], but it lifts the restriction on the degree of a strand in the dag.

Lemma 7 *Consider the execution of a dynamic multithreaded computation A by a randomized work-stealing scheduler on an ideal parallel computer with P processors. For any $\epsilon > 0$, with probability $1-\epsilon$, the execution terminates with at most $O(P(\mathrm{Span}(A_\pi)+\lg(1/\epsilon)))$ dollars in the* Steal *bucket, and the expected number of dollars in this bucket is $O(P \cdot \mathrm{Span}(A_\pi))$.*

Proof. Lemma 5 shows that, if at least $4PR$ steal attempts occur, then some delay sequence (p, R, Π) must occur. Consider a particular delay sequence (p, R, Π) with delaying path $p = \langle u_1, u_2, \ldots, u_L\rangle$ and permutation $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \ldots, \pi_L, \pi'_L)$. The construction of deque dependencies in A_ϕ ensures that $L \leq 2 \cdot \mathrm{Span}(A_\pi)$.

Such a sequence occurs if, for $i = 1, 2, \ldots, L$, each strand u_i is critical throughout all rounds in π_i. Lemma 6 shows that the probability that all π_i rounds in the ith group occur while a given strand u_i is critical is at most the probability that only 0 or 1 of the steal attempts initiated in the first $\pi_i - 1$ of these rounds chose u_i's worker, which is at most $e^{-2\pi_i+3}$ for $\pi_i \geq 2$. For $\pi_i < 2$, probability 1 suffices an upper bound. Moreover, because the work stealing scheduler chooses all targets of work-steal attempts independently, the probability of the delay sequence (p, R, Π) occurring satisfies the following:

$$\begin{aligned}
\Pr\{(p,R,\Pi)\text{ occurs}\} &= \prod_{1\leq i\leq L} \Pr\{\text{the rounds in } i\text{th group occur while } u_i \text{ is critical}\} \\
&\leq \prod_{1\leq i\leq L;\ \pi_i\geq 2} e^{-2\pi_i+3} \\
&\leq \exp\left[-2\left(\sum_{1\leq i\leq L;\ \pi_i\geq 2} \pi_i\right) + 3L\right] \\
&= \exp\left[-2\left(\sum_{1\leq i\leq L} \pi_i - \sum_{1\leq i\leq L;\ \pi_i< 2} \pi_i\right) + 3L\right] \\
&\leq e^{-2((R-L)-L)+3L} \\
&= e^{-2R+7L}\ ,
\end{aligned}$$

where the final inequality follows from Inequality (3.1).

To bound the probability of any delay sequence (p, R, Π) occurring, we multiply the number of possible delay sequences and by the probability that a particular delay sequence occurs. Because the maximum out-degree of any strand in A_ϕ is 2, there are at most $2^{2\cdot\mathrm{Span}(A_\pi)}$ distinct maximal paths in A_ϕ, and there are hence at most $2^{2\cdot\mathrm{Span}(A_\pi)}$ distinct choices for a delaying path p. There are at most

$$\binom{2L+R}{R} \leq \binom{4\cdot\mathrm{Span}(A_\pi)+R}{R}$$

ways to choose Π, because Π partitions R into at most $2\cdot\mathrm{Span}(A_\pi)$ pieces. We just observed that a given delay sequence has at most an $e^{-2R+14\cdot\mathrm{Span}(A_\pi)}$ chance of occurring. Multiplying these three factors together bounds the probability than any delay sequence (p, R, Π) occurs to be

$$2^{2\cdot\mathrm{Span}(A_\pi)} \cdot \binom{4\cdot\mathrm{Span}(A_\pi)+R}{R} \cdot e^{-2R+14\cdot\mathrm{Span}(A_\pi)}\ ,$$

which is at most ϵ for $R = c \cdot \mathrm{Span}(A_\pi) + \lg(1/\epsilon)$, where c is a sufficiently large positive constant. The probability that at least $4PR = \Theta(P(\mathrm{Span}(A_\pi) + \lg(1/\epsilon)))$ steal attempts occur is therefore at most ϵ. The expectation bound follows because the tail of the distribution decreases exponentially. □

Next, we bound the number of dollars in the WAIT-STEAL bucket by observing that Blumofe and Leiserson's analysis applies.

Lemma 8 *Consider the execution of a dynamic multithreaded computation A on an ideal parallel computer with P processors using a randomized work-stealing scheduler. For any $\epsilon > 0$, with probability at least $1 - \epsilon$, the number of dollars in the* WAIT-STEAL *bucket is at most a constant times the number of dollars in the* STEAL *bucket plus $O(P \lg P + P \lg(1/\epsilon))$, and the expected number of dollars in the* WAIT-STEAL *bucket is at most the number in the* STEAL *bucket.*

PROOF. The proof follows from the analysis of Blumofe and Leiserson [58, Lemma 6]. □

The next lemma bounds the work required to perform all eliminations using the locking protocol in Figure 3-10 in order to bound the number of dollars in the WAIT-LOCK bucket.

Lemma 9 *Consider the execution of a dynamic multithreaded computation A on an ideal parallel computer with P processors using a randomized work-stealing scheduler. If M successful steals occur during the execution of A, then the expected number of dollars in the* WAIT-LOCK *bucket is $O(M)$, and with probability at least $1-\epsilon$, at most $O(M+\lg P+\lg(1/\epsilon))$ dollars end up in the* WAIT-LOCK *bucket.*

PROOF. We apply the analysis of the locking protocol presented in [144] to the locking protocol presented in Figure 3-10. Because lines 58–71 in Figure 3-10 all require $\Theta(1)$ work, each abstract lock is held for $\Theta(1)$ time. Because only two workers can compete for any given abstract lock simultaneously, assuming linear waiting on locks [76], the total amount of time workers spend waiting for workers holding two abstract locks is at most proportional the number M of successful steals. We only need, therefore, to analyze the time workers that are holding only one abstract lock spend waiting for their second abstract lock.

Consider the eliminations attempts performed by a given worker w, and assume that w performed m elimination attempts and hence $2m$ abstract lock acquisitions. Consider the steal tree at the time of the ith abstract lock acquisition by w, when w is trying to eliminate frame F. Each worker x that is trying to eliminate some other frame $F_x \neq F$ in the steal tree at the same time creates an arrow across F_x oriented in the direction from the first edge x abstractly locks to the second. These arrows create directed paths in the tree that represent chains of dependencies on a given abstract lock, and the delay for w's ith lock acquisition can be at most the length of such a directed path starting at the edge that w is abstractly locking. Because the orientation of lock acquisition along this path is fixed, and each pair of acquisitions is correctly oriented with probability 1/2, the waiting time for F acquiring one of its locks can be bounded by a geometric distribution:

$$\Pr\{w \text{ waits for at least } k \text{ elimination attempts}\} \leq 2^{-k-1} .$$

We compute a bound on the total time Δ for all $2m$ abstract lock acquisitions by worker w. First, we must prove that the ith abstract lock acquisition by some worker w is independent of the time for the jth abstract lock acquisition for $j > i$.

To prove this independence result, we argue that, for two workers w and v, we have $\Pr\{v \text{ delays } w_j \mid v \text{ delays } w_i\} = \Pr\{v \text{ delays } w_j \mid v \text{ does not delay } w_i\} = \Pr\{v \text{ delays } w_j\}$, where w_i and w_j are w's ith and jth lock acquisitions, respectively. We consider each of these cases separately. First, suppose that worker v delays acquisition w_i. After w_i, worker v has succeeded in acquiring and releasing its abstract locks, and all lock acquisitions in the directed path from w's lock acquisition to v's have also succeeded. For v to delay acquisition w_j, a new directed path of dependencies from w to v must occur. Each edge in that path is oriented correctly with a 1/2 probability, regardless of any previous interaction between v and w. Similarly, suppose that v does not delay w_i. For v to delay w_j, a chain of dependencies must form from one of w's abstract locks to one of v's abstract locks after w_i completes. Forming such a dependency chain requires every edge in the chain to be correctly oriented, which occurs with a 1/2 probability per edge regardless of the fact that v did not delay w_i. We conclude that $\Pr\{v \text{ delays } w_j \mid v \text{ delays } w_i\} = \Pr\{v \text{ delays } w_j \mid v \text{ does not delay } w_i\} = \Pr\{v \text{ delays } w_j\}$.

For all workers $v \neq w$, the probability that v delays acquisition w_j is therefore independent of whether v delays acquisition w_i, and every lock acquisition by some worker is independent of all previous acquisitions. The probability that the $2m$ acquisitions take time longer than Δ elimination attempts is therefore at most

$$\begin{aligned}\binom{\Delta}{2m}2^{-\Delta} &\leq \left(\frac{e\Delta}{2m}\right)^{2m} 2^{-\Delta} \\ &\leq \epsilon'/P\end{aligned}$$

by choosing $\Delta = c(m + \lg(1/\epsilon'))$, where c is a sufficiently large constant greater than 1.

Next, we bound the number of elimination attempts that occur. Because each successful steal creates a frame in the steal tree that must be eliminated, the number of elimination attempts is at least as large as the number M of successful steals. Each successful elimination can force up to 2 other frames to repeat this protocol, thereby increasing the number of elimination attempts by at most 2. The total number of elimination attempts is therefore at most $3M$.

Because there are at most P workers, the total number of dollars in the WAIT-LOCK bucket is $O(M + \lg P + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$ (letting $\epsilon = \epsilon'/P$). The expectation bound follows directly. □

The next lemma combines these bounds on the dollars within each bucket to yield the expected running time of a multithreaded computation A.

Lemma 10 *Consider the execution of a dynamic multithreaded computation A on an ideal parallel computer with P processors using a randomized work-stealing scheduler. The expected running time of A is $T_P(A) \leq \text{Work}(A_\pi)/P + O(\text{Span}(A_\pi))$, and for any $\epsilon > 0$, with probability at least $1-\epsilon$, the execution time on P processors is $\text{Work}(A_\pi)/P + O(\text{Span}(A_\pi) + \lg P + \lg(1/\epsilon))$.*

PROOF. Lemma 7 shows that the number successful steal attempts is $O(P \cdot \text{Span}(A_\pi))$ in expectation and $O(P(\text{Span}(A_\pi) + \lg(1/\epsilon)))$ with probability at least $1 - \epsilon$. Lemma 9 shows that the expected number of dollars in the WAIT-LOCK bucket is $O(P \cdot \text{Span}(A_\pi))$. Summing over the number of dollars in the remaining buckets argued in Lemmas 4 and 8 and dividing by P gives the stated bounds. □

Relating the performance and user dags

We conclude the proof with two more lemmas to bound the work and span of the performance dag in terms of the work and span of the user dag.

Lemma 11 *Consider the execution of a dynamic multithreaded computation A with user dag A_ν and performance dag A_π, and let τ be the worst-case cost of any* Create-Identity *or* Reduce *operation for the given input. Then we have* $\mathrm{Span}(A_\pi) = O(\tau \cdot \mathrm{Span}(A_\nu))$.

Proof. Each successful steal in the execution of A can force one call to Create-Identity, which creates a nonempty view that must later be reduced using a Reduce operation. Therefore, at most one Reduce operation can occur per successful steal, and at most one reduce strand can occur in the performance dag for each steal. Each spawn in A_ν provides an opportunity for a steal to occur. Consequently, each spawn operation in A can increase the size of the dag by 2τ in the worst case.

Consider a critical path p_π in A_π, and let p_ν be the corresponding path in A_ν, that is, the path in A_ν made up exactly of the user strands in p_π. Suppose that k steals occur along p_ν. The length of the path p_π in A_π is then at most $2k\tau + |p_\nu| \leq 2\tau \cdot \mathrm{Span}(A_\nu) + |p_\nu| \leq 3\tau \cdot \mathrm{Span}(A_\nu)$, which implies that $\mathrm{Span}(A_\pi) = O(\tau \cdot \mathrm{Span}(A_\nu))$. □

Lemma 12 *Consider the execution of a dynamic multithreaded computation A with user dag A_ν, and let τ be the worst-case cost of any* Create-Identity *or* Reduce *operation for the given input. For any $\epsilon > 0$, with probability $1 - \epsilon$, the work of A_π is* $\mathrm{Work}(A_\nu) + O(\tau^2 P \cdot \mathrm{Span}(A_\nu) + \tau P \lg(1/\epsilon))$*, and the expected work of A_π is* $\mathrm{Work}(A_\nu) + O(\tau^2 P \cdot \mathrm{Span}(A_\nu))$.

Proof. The work in A_π equals the work in A_ν plus the total work of all runtime strands in A. Because at most one call to Create-Identity and one call to Reduce can occur for each successful steal in the execution of A, the number of reduce and init strands in A_π is bounded above by the number of successful steals M. Each reduce or init strand incurs at most τ work to execute, meaning that the total work in reduce and init strands is τM. The lemma follows from applying Lemmas 7 and 11 to bound M. □

We now prove Inequality (2.5), which bounds the running time of a computation whose nondeterminism arises from reducers.

Theorem 13 *Consider the execution of a computation A on a parallel computer with P processors using a work-stealing scheduler. Let A_ν be the user dag of A. The expected running time of A is* $T_P(A) \leq \mathrm{Work}(A_\nu)/P + O(\tau^2 \cdot \mathrm{Span}(A_\nu))$*, and for any $\epsilon > 0$, with probability $1 - \epsilon$, the execution time is* $T_P(A) \leq \mathrm{Work}(A_\nu)/P + O(\tau^2 \cdot \mathrm{Span}(A_\nu) + \lg P + \tau \lg(1/\epsilon))$.

Proof. The proof follows from combining Lemmas 10, 11 and 12. □

3.7 Analyzing PBFS

This section applies the results of Section 3.6 to bound the expected running time of PBFS. Specifically, we analyze the locking version of PBFS, for which a vertex is added at most once to any bag. We analyze the work and span of the user dag for PBFS. Using Theorem 13 to combine these bounds with performance bounds on the bag data structure methods, we bound the expected running time of PBFS. In particular, for a connected graph $G = (V, E)$

with bounded maximum out-degree, PBFS executes in $T_P \leq \Theta(V+E)/P + O(D \lg^3(V/D))$ expected time.

We first bound the work and span of the user dag for PBFS.

Lemma 14 *Consider an execution of the locking version of PBFS on a connected graph $G = (V, E)$ with diameter D, and let Δ denote the maximum out-degree of any vertex in V. The total work in PBFS's user dag is $\Theta(V+E)$, and the total span of PBFS's user dag is $O(D \lg(V/D) + D \lg \Delta)$, where Δ is the maximum out-degree of any vertex in V.*

PROOF. We measure work and span of PBFS by studying its pseudocode in Figures 3-3 and 3-4.

To measure the work of PBFS, consider evaluating the dth layer V_d of G. PBFS evaluates every vertex $v \in V_d$ exactly once, and PBFS checks every successor vertex u of v exactly once. In the locking version of PBFS, each vertex u is assigned its distance exactly once and added to the bag for layer V_{d+1} exactly once. Because these properties hold for all layers of G, the total work for this portion of PBFS is $\Theta(V+E)$.

PBFS performs additional bookkeeping work in order to store the vertices within a bag. In particular, PBFS performs additional work to create a bag for each layer, to subdivide a bag into GRAINSIZE pieces, and to insert vertices into a bag. We shall show that this additional bookkeeping work totals to $\Theta(V)$ work, implying that the total work of PBFS is $\Theta(V+E)$.

To create a bag for each layer, PBFS calls BAG-CREATE once per layer, which incurs $\Theta(D)$ total work across all D layers.

To subdivide a bag into GRAINSIZE pieces, PBFS first subdivides a bag into pennants (lines 21–23) and then recursively splits each pennant (lines 31–34), which requires $\Theta(V_d)$ work per layer and $\Theta(V)$ work over all layers.

The total time PBFS spends executing BAG-INSERT depends on the parallel execution of PBFS. Since a steal resets the contents of a bag for subsequent update operations, the maximum running time of BAG-INSERT depends on the steals that occur. Each steal can only decrease the work of a subsequent BAG-INSERT, and therefore the amortized running time of $\Theta(1)$ for each BAG-INSERT still applies. Because BAG-INSERT is called once per vertex, PBFS spends $\Theta(V)$ work total executing BAG-INSERT.

The span of PBFS is dominated by the sum of the spans for processing each layer of G. The span of a PROCESS-LAYER call on the dth layer is at most the span of the **for** loop on lines 21–23 — $O(\lg V_d)$ — plus the maximum span of any PROCESS-PENNANT call on line 23. The total span for any such PROCESS-PENNANT call is the sum of the span to recursively split a pennant in lines 31–33, which is $O(\lg V_d)$; plus the span to process a single vertex; plus the span to sync all spawned children, which is also $O(\lg V_d)$. The span of processing a single vertex is dominated by lines 26 and 28.4, which have a total span of $O(\lg \Delta + \lg V_{d+1})$. The span of processing the dth layer of G is therefore $O(\lg V_d + \lg V_{d+1} + \lg \Delta)$, making the total span of PBFS $O(D \lg(V/D) + D \lg \Delta)$. □

We now bound the expected running time of PBFS.

Theorem 15 *Consider the parallel execution of PBFS on a connected graph $G = (V, E)$ with diameter D running on a parallel computer with P processors using a work-stealing scheduler. The expected running time of the locking version of PBFS is $T_P(\text{PBFS}) \leq \Theta(V+E)/P + O(D \lg^2(V/D)(\lg(V/D) + \lg \Delta))$, where Δ is the maximum out-degree of any vertex in V.*

PROOF. To maximize the cost of all CREATE-IDENTITY and REDUCE operations in PBFS, the worst-case cost of each of these operations must be $O(\lg(V/D))$. Combining Lemma 14 with Theorem 13, where $\tau = O(\lg(V/D))$, we get $T_P \leq \Theta(V + E)/P + O(D \lg^2(V/D)(\lg(V/D) + \lg \Delta))$. □

3.8 Conclusion

Thread-local storage [372], or TLS, presents an alternative to bag reducers for implementing the layer sets in a parallel breadth-first search. The bag reducer allows PBFS to write the vertices of a layer in a single data structure in parallel and later efficiently traverse them in parallel. As an alternative to bags, each of the P workers could store the vertices it encounters into a vector within its own TLS, thereby avoiding races. The set of elements in the P vectors could then be walked in parallel using divide-and-conquer. Such a structure appears simple to implement and practically efficient, since it avoids merging sets.

Despite the simplicity of the TLS solution, reducer-based solutions exhibit some advantages over TLS solutions, including a simple programming model that supports a scientific approach to reasoning about fast multicore software. First, reducers provide a processor-oblivious alternative to TLS, enhancing portability and simplifying reasoning about how performance scales. Second, reducers allow a function to be instantiated multiple times in parallel without interference. To support simultaneous running of functions that use TLS, the programmer must manually ensure that the TLS regions used by the functions are disjoint. Third, reducers require only a monoid — associativity and an identity — to ensure correctness, whereas TLS also requires commutativity. The correctness of some applications, including BFS, is not compromised by allowing commutative updates to its shared data structure. Without commutativity, an application cannot easily use TLS, whereas reducers seem to be good whether commutativity is allowed or not. Finally, whereas TLS makes the nondeterminism visible to the programmer, reducers encapsulate nondeterminism. In particular, reducers hide the particular nondeterministic manner in which associativity is resolved, thereby allowing the programmer to assume specific semantic guarantees at well-defined points in the computation. This encapsulation of nondeterminism simplifies the task of reasoning about the program's correctness compared to a TLS solution.

Nondeterminism can wreak havoc on the ability to reason about programs, to test their correctness, and to ascertain their performance, but it also can provide opportunities for additional parallelism. Well-structured linguistic support for encapsulating nondeterminism can allow parallel programmers to enjoy the benefits of nondeterminism without suffering unduly from the inevitable complications that nondeterminism engenders. Reducers provide an effective way to encapsulate nondeterminism. We view it as an open question whether a semantics exists for TLS that would encapsulate nondeterminism while providing a potentially more efficient implementation in situations where commutativity is allowed.

Chapter 4

Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling

This chapter presents the Prism chromatic-scheduling algorithm for executing dynamic data-graph computations [213]. This work was conducted in collaboration with Tim Kaler, William Hasenplaugh, and Charles E. Leiserson.

4.1 Introduction

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

A data-graph computation is an algorithm that performs "local" updates on the vertices of a data graph, taking as input data associated with a vertex and its neighbors. Several software systems have been implemented to support parallel data-graph computations, including GraphLab [264, 265], Pregel [269], Galois [298, 299], PowerGraph [165], Ligra[1] [358, 361], and GraphChi [228]. These systems can support "complex" data-graph computations, in which data can be associated with edges as well as vertices and updating a vertex v can modify any data associated with v, v's incident edges, and the vertices adjacent to v. For ease in discussing chromatic scheduling, however, we shall principally restrict ourselves to "simple" data-graph computations (which correspond to "edge-consistent" computations in GraphLab), although most of our results straightforwardly extend to more complex models. Indeed, six out of the seven GraphLab applications described in [264, 265] are simple data-graph computations.

In a data-graph computation, updates to vertices proceed in ***rounds***, where each vertex can be updated at most once per round. In a ***static*** data-graph computation, the ***activation set*** Q_r of vertices updated in a round r — the set of ***active*** vertices — is determined a

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

priori. Often, a static data-graph computation updates every vertex in each round. Static data-graph computations include Gibbs sampling [155, 156], iterative graph coloring [104], and n-body problems such as the fluidanimate PARSEC benchmark [45].

We shall be interested in ***dynamic*** data-graph computations, where the activation set changes round by round. Dynamic data-graph computations include the Google PageRank algorithm [68], loopy belief propagation [292, 313], coordinate descent [110], co-EM [300], alternating least-squares [185], singular-value decomposition [164], and matrix factorization [391].

We formalize the computational model as follows. Let $G = (V, E)$ be a data graph. We shall denote the ***neighbors***, or ***adjacent vertices***, of a vertex $v \in V$ by $\text{Adj}[v] = \{u \in V : (u, v) \in E\}$. The ***degree*** of v is thus $\deg(v) = |\text{Adj}[v]|$, and the ***degree*** of G is $\deg(G) = \max\{\deg(v) : v \in V\}$. A ***(simple) dynamic data-graph computation*** is a triple $\langle G, f, Q_0 \rangle$, where

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

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

The advantage of dynamic over static data-graph computations is that they avoid performing many unnecessary updates. Studies in the literature [264, 265] show that dynamic execution can enhance the practical performance of many applications. We confirmed these findings by implementing static and dynamic versions of several data-graph computations. The results for a PageRank algorithm on a power-law graph of 1 million vertices and 10 million edges were typical. The static computation performed approximately 15 million updates, whereas the dynamic version performed less than half that number of updates.

A serial reference implementation

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

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

We can analyze the time SERIAL-DDGC takes to execute one round r of the data-graph

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

Figure 4-1: Pseudocode for a serial algorithm to execute a data-graph computation $\langle G, f, Q_0 \rangle$. SERIAL-DDGC takes as input a data graph G and an update function f. The computation maintains a FIFO queue Q of activated vertices that have yet to be updated and sentinel values NIL, each of which demarcates the end of a round. An update $S = f(v)$ returns the set $S \subseteq \text{Adj}[v]$ of vertices activated by that update. Each vertex $u \in S$ is added to Q if it is not currently scheduled for a future update.

computation $\langle G, f, Q_0 \rangle$. Define the ***size*** of an activation set Q_r as

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

The size of Q_r is asymptotically the space needed to store all the vertices in Q_r and their incident edges using a standard sparse-graph representation, such as compressed-sparse-rows (CSR) format [374]. For example, if $Q_0 = V$, we have $\textit{size}(Q_0) = |V| + 2|E|$ by the handshaking lemma [100, p. 1172–3]. Let us make the reasonable assumption that the time to execute $f(v)$ serially is proportional to $\deg(v)$. If we implement the queue as a dynamic (resizable) table [100, Section 17.4], then line 14 executes in $\Theta(1)$ amortized time. Of course, a linked list would suffice to append operations in $\Theta(1)$ time, but would not allow for convenient subsequent parallel iteration over its elements. All other operations in the **for** loop on lines 12–14 take $\Theta(1)$ time, and thus all vertices activated by executing $f(v)$ are examined in $\Theta(\deg(v))$ time. The total time spent updating the vertices in Q_r is therefore $\Theta(Q_r + \sum_{v \in Q_r} \deg(v)) = \Theta(\textit{size}(Q_r))$, which is ***linear*** time: time proportional to the storage requirements for the vertices in Q_r and their incident edges.

Parallelizing dynamic data-graph computations

The salient challenge in parallelizing data-graph computations is to deal effectively with races between logically parallel updates that read and write common data. Two updates in a data-graph computation ***conflict*** if executing them in parallel produces a determinacy race. A parallel scheduler must manage or avoid conflicting updates to execute a data-graph computation correctly and deterministically.

The standard approach to preventing races associates a mutual-exclusion lock with each vertex of the data graph to ensure that an update on a vertex v does not proceed until all locks on v and v's neighbors have been acquired. Although this locking strategy prevents

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

Figure 4-2: Comparison of dynamic data-graph schedulers on seven application benchmarks. All running times are in seconds and were calculated by taking the median 12-core execution time of 5 runs on an Intel Xeon X5650 with hyperthreading disabled. The running times of Prism and Prism-R include the time used to color the input graph. PR/G and PR/L run a PageRank algorithm on the web-Google [252] and soc-LiveJournal [27] graphs, respectively. ID/2000 and ID/4000 run an image denoise algorithm to remove Gaussian noise from 2D grayscale images of dimension 2000 by 2000 and 4000 by 4000. FBP/C1 and FBP/C3 perform belief propagation on a factor graph provided by the cora-1 and cora-3 datasets [276, 362]. ALS/N runs an alternating least squares algorithm on the NPIC-500 dataset [284].

data races, it can incur substantial overhead from lock acquisition and contention, hurting application performance, especially when update functions are simple. Moreover, because runtime happenstance can determine the order in which two logically parallel updates acquire locks, the data-graph computation can behave nondeterministically: different runs on the same inputs can produce different results. Without repeatability, performance engineers face a significant challenge in understanding what a program does and in finding and removing programming bugs.

A known alternative to using locks is ***chromatic scheduling*** [2,43,264], which schedules a data-graph computation based on a coloring of the data-graph computation's ***conflict graph*** — a graph with an edge between two vertices if updating them in parallel would produce a determinacy race. For a simple data-graph computation, the conflict graph is simply the data graph itself with undirected edges. The idea behind chromatic scheduling is fairly simple. Chromatic scheduling begins by computing a ***(vertex) coloring*** of the conflict graph — an assignment of colors to the vertices such that no two adjacent vertices share the same color. Since no edge in the conflict graph connects two vertices of the same color, updates on all vertices of a given color can execute in parallel without producing races. To execute a round of a data-graph computation, the set of activated vertices Q is partitioned into χ ***color sets*** — subsets of Q containing vertices of a single color. Updates are applied to vertices in Q by serially stepping through each color set and updating all vertices within a color set in parallel. Indeed, the special case where the active set $Q = V$ is the entire vertex set of the graph (i.e., a static data-graph computation) can be executed using chromatic scheduling using Distributed GraphLab [264]. The result of a data-graph computation executed using chromatic scheduling is equivalent to that of a slightly modified version of Serial-DDGC that starts each round (immediately before line 9 of Figure 4-1) by sorting the vertices within its queue by color.

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

To date, chromatic scheduling has been applied to static data-graph computations [264], but not to dynamic data-graph computations. This chapter addresses the question of how to perform chromatic scheduling efficiently when the activation set changes on the fly, necessitating a data structure for maintaining dynamic sets of vertices in parallel.

Contributions

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

We analyze the performance of PRISM using work-span analysis. We shall make the reasonable assumption that a single update $f(v)$ executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span.[2] Under this assumption, on a Δ-degree data graph G colored using χ colors, PRISM executes the updates on the vertices in the activation set Q_r of a round r on P processors in $O(\mathit{size}(Q_r) + P)$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta) + \lg P)$ span.

The "price of determinism" incurred by using chromatic scheduling instead of the more common locking strategy appears to be negative for real-world applications. This discovery is perhaps surprising since it would seem to be strictly harder to guarantee that the computation behave deterministically than to allow for nondeterministic behaviors. Nevertheless, as Figure 4-2 indicates, on seven application benchmarks, PRISM executes 1.2 to 2.1 times faster than GraphLab's comparable, but nondeterministic, locking strategy, which we call RRLOCKS. This performance gap is not due solely to superior engineering or load balancing. A similar performance overhead is observed in a comparably engineered lock-based scheduling algorithm, CILK+LOCKS. PRISM outperforms CILK+LOCKS on each of the 7 application benchmarks and is on average (geometric mean) 1.4 times faster.

Prism offers several benefits to performance-engineering dynamic data-graph computations. By executing these computations deterministically, Prism allows programmers to reason about the behavior of these computations based on a serial execution. Prism also offers theoretical guarantees to execute dynamic data-graph computations efficiently on parallel processors, which programmers can use to investigate the scalability of these computations. Finally, our Prism implementation bears out these theoretical performance guarantees in practice, allowing programmers to convert their theoretical predictions into efficient software.

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

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

[3]Deterministic Galois [299] has added support for deterministic execution of dynamic data-graph com-

effect of our scheduling algorithms. Thus, the empirical comparisons in this chapter are apples-to-apples comparisons of scheduling strategies, not competitive comparisons with other systems.

PRISM behaves deterministically as long as every update is ***pure***: it modifies no data except for that associated with its target vertex. This assumption precludes the update function from modifying global variables to aggregate or collect values. To support this common use pattern, we describe an extension to PRISM, called PRISM-R, which executes dynamic data-graph computations deterministically even when updates modify global variables using associative operations. PRISM-R replaces each "multibag" that PRISM uses with a "multivector," maintaining color sets whose contents are ordered deterministically. PRISM-R executes in the same theoretical bounds as PRISM, but its implementation is more involved. Empirically, PRISM-R is on average (geometric mean) only 1.07 times slower than PRISM and outperforms CILK+LOCKS on all but one of the seven application benchmarks.

Outline

The remainder of this chapter is organized as follows. Section 4.2 describes PRISM, the chromatic-scheduling algorithm for dynamic data-graph computations. Section 4.3 describes the "multibag" data structure PRISM uses to represent its color sets. Section 4.4 presents our theoretical analysis of PRISM. Section 4.5 describes a Cilk Plus [196] implementation of PRISM and presents empirical results measuring this implementation's performance on seven application benchmarks. Section 4.6 describes PRISM-R which executes dynamic data-graph computations deterministically even when update functions modify global variables using associative operations. Section 4.7 describes and analyzes the "multivector" data structure PRISM-R uses to represent deterministically ordered color sets. Section 4.8 analyzes PRISM-R both theoretically, using work-span analysis, and empirically. Section 4.9 offers some concluding remarks.

4.2 The PRISM algorithm

This section presents PRISM, a chromatic-scheduling algorithm for executing dynamic data-graph computations deterministically. We describe how PRISM differs from the serial algorithm, SERIAL-DDGC, in Section 4.1, including how it maintains activation sets that are partitioned by color using the "multibag" data structure.

Figure 4-3 presents the pseudocode for PRISM, which differs from SERIAL-DDGC (shown in Figure 4-1) in two main ways: the use of a "multibag" data structure to implement Q, and the call to COLOR-GRAPH on line 15 to color the data graph.

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

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

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

putations by recursively removing and executing independent sets of vertices. However, their algorithm is not work-efficient and, as a result, is much slower than the nondeterministic version.

```
PRISM($G, f, Q_0$)
15  $\chi$ = COLOR-GRAPH($G$)
16  $r = 0$
17  $Q = Q_0$
18  while $Q \neq \emptyset$
19      $\mathcal{C}$ = MB-COLLECT($Q$)
20      for $C \in \mathcal{C}$
21          parallel for $v \in C$
22              $active[v]$ = FALSE
23              $S = f(v)$
24              parallel for $u \in S$
25                  if CAS($active[u]$, FALSE, TRUE)
26                      MB-INSERT($Q, u, color[u]$)
27      $r = r + 1$
```

```
CAS($current, test, value$)
28  begin atomic
29  if $current == test$
30      $current = value$
31      return TRUE
32  else
33      return FALSE
34  end atomic
```

Figure 4-3: Pseudocode for PRISM, including the compare-and-swap synchronization primitive CAS. The procedure PRISM takes as input a data graph G, an update function f, and an initial activation set Q_0. The procedure COLOR-GRAPH colors a given graph and returns the number of colors it used. The procedures MB-COLLECT and MB-INSERT operate the multibag Q to maintain activation sets for PRISM. The variable r tracks the number of rounds executed.

multibag.

PRISM calls COLOR-GRAPH on line 15 to color the given data graph $G = (V, E)$ and obtain the number χ of colors used. Although it is NP-complete to find an ***optimal*** coloring of a graph [150] — a coloring that uses the smallest possible number of colors — an optimal coloring is not necessary for PRISM to perform well, as long as the data graph is colored deterministically, in parallel,[4] and with sufficiently few colors in practice. Many parallel coloring algorithms exist that satisfy the needs of PRISM (see, for example, [13, 32, 162, 163, 175, 209, 226, 227, 254, 382]). Our implementation of PRISM uses a multicore variant of the Jones and Plassmann algorithm [209] that produces a deterministic $(\Delta+1)$-coloring of a Δ-degree graph $G = (V, E)$ in linear work and $O(\lg V + \lg \Delta \cdot \min\{\sqrt{E}, \Delta + \lg \Delta \lg V / \lg \lg V\})$ span. Chapter 5 describes this coloring algorithm in detail.

Let us see how PRISM uses chromatic scheduling to execute a dynamic data-graph computation $\langle G, f, Q_0 \rangle$. After line 15 colors G, line 17 initializes the multibag Q with the initial activation set Q_0, and then the **while** loop on lines 18–27 executes the rounds of the data-graph computation. At the start of each round, line 19 collects the nonempty bags $\mathcal{C}$ from Q, which correspond to the nonempty color sets for the round. Lines 20–26 iterate through the color sets $C \in \mathcal{C}$ sequentially, and the **parallel for** loop on lines 21–26 processes the vertices of each C in parallel. For each vertex $v \in C$, line 23 performs the update $S = f(v)$, which returns a set S of activated vertices, and lines 24–26 insert into Q the vertices in S that were not already active before the update.

Although a vertex u can be activated by multiple neighbors, it must only be updated at most once during a round. PRISM enforces this constraint[5] by using the atomic ***compare-and-swap*** operator [183, p. 480], CAS, which is available as a synchronization primitive on most machines. The definition of CAS is given on lines 28–34 in Figure 4-3. Lines

[4]If the data-graph computation performs sufficiently many updates, a serial $\Theta(V + E)$-work greedy coloring algorithm, such as that introduced by Welsh and Powell [403], can suffice as well, since the time to color the graph would be sufficiently amortized against the work performed.

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

24–26 use the CAS primitive to activate each vertex $u \in S$ by first atomically setting $active[u] = \text{TRUE}$, and then calling MB-Insert if $active[u]$ was previously FALSE. Thus, each vertex is inserted into Q at most once during a round.

Design considerations for the implementation of multibags

The theoretical performance of Prism depends upon the properties of the multibag data structure. In particular, the multibag is carefully designed to ensure that Prism is ***work-efficient*** — that is, it performs the same asymptotic work as the serial algorithm Serial-DDGC in Figure 4-1. Before examining the design of the multibag in Section 4.3, let us first explore why maintaining active color sets in Prism in a work-efficient manner is tricky. Specifically, we shall consider two alternative strategies: bit vectors and an array of worker-local queues.

The bit-vector approach avoids the multibag altogether and simply manages activation sets using the bit vector *active* already used by Prism. Recall that if $active[i]$ is TRUE, then the vertex $v_i \in V$ indexed by i is active. Suppose that *active* were the only data structure. To iterate over all activated vertices of color k, a **parallel for** could scan through *active*, updating the vertex v_i whenever $active[i]$ is TRUE and $color[i]$ is k. This scheme requires $\Omega(V\chi)$ work per round of the computation, where χ is the number of colors returned by Color-Graph in line 15 of Figure 4-3, since the entire bit vector must be scanned χ times each round. At the cost of additional preprocessing, *active* could be organized such that vertices of the same color are assigned contiguous indexes. Even with this optimization, however, scanning *active* requires $\Omega(V)$ work each round, which is not work-efficient for dynamic computations that activate only a sparse subset of the vertices each round.

An alternative strategy that one might consider is to represent the active color sets using an array of worker-local queues. A straightforward implementation of this approach, however, is also not work-efficient. For a dynamic data-graph computation using χ colors and P processors, a total of $P\chi$ worker-local queues would be needed to maintain the set of active vertices, and $\Omega(P\chi)$ work would be required to collect all nonempty queues. As we shall see in Section 4.3, however, by using a carefully designed data structure to manage worker-local queues, we can obtain a work-efficient data structure for maintaining color sets.

4.3 The multibag data structure

This section presents the multibag data structure employed by Prism. The multibag uses worker-local sparse accumulators [160] and an efficient parallel collection operation. We describe how the MB-Insert and MB-Collect operations are implemented, and we analyze them using work-span analysis. When used in a P-processor execution of a parallel program, a multibag Q of χ bags storing n elements supports MB-Insert in $\Theta(1)$ worst-case time and MB-Collect in $O(n+\chi+P)$ work and $O(\lg n+\chi+\lg P)$ span. Such a multibag uses $O(P\chi+n)$ space.

Our implementation of Prism assumes the existence of a runtime-system-provided function Get-Worker-ID that returns an integer from 0 to $P-1$ that uniquely identifies the worker executing the current strand in $\Theta(1)$ time. Other strategies for implementing worker-local storage exist that are comparable to the strategy outlined here.

A ***sparse accumulator (SPA)*** [160] implements an array that supports lazy initialization of its elements. A SPA T contains a sparsely populated array $T.array$ of elements and

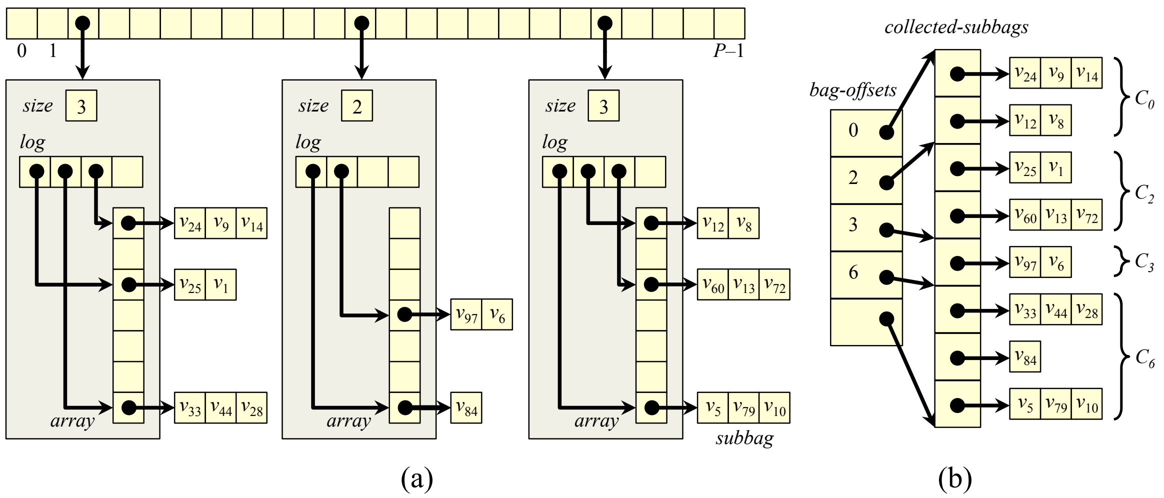


Figure 4-4: Illustration of the multibag data structure. **(a)** A multibag containing 19 elements distributed across 4 distinct bags: $\{C_0, C_2, C_3, C_6\}$, representing vertices of colors 0, 2, 3, and 6, respectively. For each bag, each worker keeps track of its portion of its subbag in a worker-local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

a log $T.\mathit{log}$, which is a list of indices of initialized elements in $T.\mathit{array}$. To implement multibags, we shall only need the ability to create a SPA, to access an arbitrary SPA element, and to delete all elements from a SPA. For simplicity, we shall assume that an uninitialized array element in a SPA has a value of NIL. When an array element $T.\mathit{array}[i]$ is modified for the first time, the index i is appended to $T.\mathit{log}$. An appropriately designed SPA T storing $n = |T.\mathit{log}|$ elements admits the following performance properties:

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

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

Implementation of MB-Insert *and* MB-Collect

The worker-local SPA's enable a multibag Q to support parallel MB-Insert operations without creating races. Figure 4-5 shows the pseudocode for MB-Insert. When a worker p executes MB-Insert(Q, v, k), it inserts element v into the subbag $C_{k,p}$ as follows. Line 35 calls Get-Worker-ID to get worker p's identifier. Line 36 checks if subbag $C_{k,p}$ stored in $Q[p].\mathit{array}[k]$ is initialized, and if not, lines 37 and 38 initialize it. Line 39 inserts v into $Q[p].\mathit{array}[k]$.

```
MB-INSERT($Q, v, k$)
35  $p$ = GET-WORKER-ID()
36  if $Q[p].array[k] == $ NIL
37      APPEND($Q[p].log, k$)
38      $Q[p].array[k]$ = new $subbag$
39  APPEND($Q[p].array[k], v$)
```

Figure 4-5: Pseudocode for the MB-INSERT multibag operation. MB-INSERT(Q, v, k) inserts the element v into the kth bag C_k of the multibag Q.

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

Figure 4-6 sketches how MB-COLLECT converts a multibag Q stored in worker-local SPA's into the representation illustrated in Figure 4-4(b). Steps 1 and 2 create an array *collected-subbags* of nonempty subbags from the worker-local SPA's in Q. Each subbag $C_{k,p}$ in *collected-subbags* is tagged with the integer index k of its corresponding bag $C_k \in Q$. Step 3 sorts *collected-subbags* by these index tags, and Step 4 creates the *bag-offsets* array. Step 5 removes all elements from Q, thereby emptying the multibag.

Analysis of multibags

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

Lemma 16 *Executing* MB-INSERT *takes* $\Theta(1)$ *time in the worst case.*

PROOF. Consider each step of a call to MB-INSERT(Q, v, k). The GET-WORKER-ID procedure on line 35 obtains the executing worker's identifier p from the runtime system in $\Theta(1)$ time, and line 36 checks if the entry $Q[p].array[k]$ is empty in $\Theta(1)$ time. Suppose that $Q[p].log$ and each subbag in $Q[p].array$ are implemented as dynamic arrays that use a deamortized table-doubling scheme [69]. Lines 37–39 then take $\Theta(1)$ time each to append k to $Q[p].log$, create a new subbag in $Q[p].array[k]$, and append v to $Q[p].array[k]$. □

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

Lemma 17 *In a program execution on P processors, a call to* MB-COLLECT(Q) *on a multibag Q with χ bags whose contents are distributed across m distinct subbags executes in $\Theta(m + \chi + P)$ work and $\Theta(\lg m + \chi + \lg P)$ span.*

PROOF. We analyze each step of MB-COLLECT in turn. We shall use a helper procedure PREFIX-SUM(A), which computes the all-prefix sums of an array A of n integers in $\Theta(n)$ work and $\Theta(\lg n)$ span. Blelloch [49] describes an appropriate implementation of PREFIX-SUM.

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

Figure 4-6: Pseudocode for the MB-Collect multibag operation. Calling MB-Collect on a multibag Q produces a pair of arrays *collected-subbags*, which contains all nonempty subbags in Q sorted by their associated bag's index, and *bag-offsets*, which associates sets of subbags in Q with their corresponding bag.

Step 1 replaces each entry in $Q[p].log$ in each worker-local SPA $Q[p]$ with the appropriate index-subbag pair $\langle k,\ C_{k,p}\rangle$ in parallel, which requires $\Theta(m + P)$ work and $\Theta(\lg m + \lg P)$ span.

Step 2 gathers all index-subbag pairs into a single array. Suppose that each worker-local SPA $Q[p]$ is augmented with the size of $Q[p].log$, as Figure 4-4(a) illustrates. Executing Prefix-Sum on these sizes and then copying the entries of $Q[p].log$ into *collected-subbags* in parallel therefore completes Step 2 in $\Theta(m + P)$ work and $\Theta(\lg m + \lg P)$ span.

Step 3 can sort the *collected-subbags* array in $\Theta(m + \chi)$ work and $\Theta(\lg m + \chi)$ span using a variant of a parallel radix sort [54, 93, 416] as follows:

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

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

Corollary 18 *In a program execution on P processors, let Q be a multibag whose contents are distributed across m distinct subbags, and suppose that all nonempty bags in Q have indices in $0, 1, \ldots, k$. Then Q may be treated as a multibag representing k bags such that* MB-Collect(Q) *executes in $\Theta(m + k + P)$ work and $\Theta(\lg m + k + \lg P)$ span.*

Proof. Steps 1 and 2 of MB-Collect can determine the index k of the largest nonempty bag in Q at no asymptotic cost to work or span, and the subsequent steps of MB-Collect can then substitute k in place of χ. The corollary thus follows from Lemma 17. □

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

4.4 Analysis of Prism

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

The theoretical analyses presented throughout this chapter assume that concurrent reads and writes incur no overheads due to contention.

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

Theorem 19 *Suppose that* Prism *colors a Δ-degree data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0\rangle$. Then, on P processors,* Prism *executes updates on all vertices in the activation set Q_r for a round r using $\Theta(size(Q_r) + P)$ work and $O(\chi(\lg(Q_r/\chi) + \lg\Delta) + \lg P)$ span.*

Proof. Let us first analyze one iteration of the **for** loop on lines 20–26 in Prism, which performs the updates on the vertices belonging to one color set $C \in Q_r$. Consider a vertex $v \in C$. Lines 22 and 23 execute in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span. For each vertex u in the set S of vertices activated by the update $f(v)$, Lemma 16 implies that lines 25–26 execute in $\Theta(1)$ total work. The **parallel for** loop on lines 24–26 therefore executes in $\Theta(S)$ work and $\Theta(\lg S)$ span. Because $|S| \le \deg(v)$, the **parallel for** loop on lines 21–26 thus executes in $\Theta(size(C))$ work and $\Theta(\lg C + \max_{v\in C}\lg(\deg(v))) = O(\lg C + \lg\Delta)$ span.

By processing each of the χ color sets belonging to Q_r, lines 20–26 therefore executes in $\Theta(size(Q_r) + \chi)$ work and $O(\chi(\lg(Q_r/\chi) + \lg\Delta))$ span. Corollary 18 implies that line 19 executes MB-Collect in $\Theta(Q_r + \chi_r + P)$ work and $\Theta(\lg Q_r + \chi_r + \lg P)$ span where $\chi_r = \max_{v\in Q_r}\{color[v]\}$. The theorem follows since $|Q_r| + \chi_r \le size(Q_r) + 1$ □

Theoretical scalability of Prism

Dynamic data-graph computations typically run for multiple rounds until a convergence criteria is met. We will now generalize Theorem 19 to prove work and span bounds for Prism when executing a sequence of rounds.

Theorem 20 *Suppose that* Prism *colors a Δ-degree data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0\rangle$ in r rounds applying updates to the activation sets $Q_0, Q_1, \ldots, Q_{r-1}$. Define the multiset $\mathcal{U} = \biguplus_{i=0}^{r-1} Q_i$ so that $|\mathcal{U}| = \sum_{i=0}^{r-1}|Q_i|$ and $size(\mathcal{U}) = \sum_{i=0}^{r-1} size(Q_i)$, where the symbol $\biguplus$ indicates a* ***multiset sum****.*[6] *Then, on*

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

P processors, Prism *executes the data-graph computation using $\Theta(\mathit{size}(\mathcal{U}) + rP)$ work and $O(r\ \chi(\lg((\mathcal{U}/r)/\chi) + \lg\Delta) + r\lg P)$ span.*

Proof. The work bound follows directly from Theorem 19 by taking the sum of work performed in each of the r rounds of Prism. The total span of Prism is equal to the sum of each round's span, which Theorem 19 bounds by $\sum_{i=0}^{r-1} O(\chi(\lg(Q_i/\chi) + \lg\Delta) + \lg P)$. Observing that $\sum_{i=0}^{r-1} \chi\lg(Q_i/\chi) \le r\ \chi\lg((\mathcal{U}/r)/\chi)$ completes the proof. □

Given Theorem 20 we can compute the parallelism of Prism for a data-graph computation that applies a multiset $\mathcal{U}$ of updates over r rounds. The following corollary expresses the parallelism of Prism in terms of the average size of the activation sets in a sequence of rounds.

Corollary 21 *Suppose* Prism *executes a data-graph computation in r rounds during which it applies a multiset $\mathcal{U}$ of updates. Define the average number of updates per round $U_{avg} = |\mathcal{U}|/r$ and the average work per round $W_{avg} = \mathit{size}(\mathcal{U})/r$. Then* Prism *has parallelism of $\Omega(W_{avg}/(\chi(\lg(U_{avg}/\chi) + \lg\Delta)))$.*

Proof. Follows from Theorem 20 by computing the parallelism as the ratio of the work over the span and then performing substitution. □

Corollary 21 implies that Prism achieves near perfect linear parallel speedup on P processors for a graph of n vertices when the average work W_{avg} performed in each round is much larger than $P\ \chi\lg n$.

4.5 Empirical evaluation

This section explores the performance properties of Prism from an empirical perspective. We describe three experiments designed to investigate the synchronization costs, dynamic-scheduling overheads, and scalability properties of Prism. For the first experiment, on a suite of 12 benchmark graphs, Prism executed between 1.0 and 2.1 times faster than a nondeterministic locking protocol on PageRank [68], exhibiting a geometric-mean speedup of a factor of 1.5. Prism thus exhibits substantially lower synchronization costs. The second experiment shows that, compared with static scheduling, the slowdown that Prism incurs for dynamic scheduling using multibags is only about 1.16 when all vertices are activated in every round. This experiment shows that Prism can be effective even for relatively densely activated graphs. The third experiment shows that Prism scales well and is relatively insensitive to the number of colors needed to color the conflict graph, as long as there is sufficient parallelism.

The results presented here include data for Prism-R. We will discuss this data in Section 4.8, after presenting Prism-R.

Experimental setup

Figure 4-7 summarizes the technical specifications of the machine used for all of the experiments. All code was compiled using `-O3` optimizations.

As a platform for our experiments, we implemented a new parallel execution engine within GraphLab [265] that uses Intel Cilk Plus [196] to expose parallelism. The new execution engine and all of our scheduling algorithms were designed to be compatible with the original GraphLab API in order to facilitate a fair evaluation of the relative merits of

CPU	Intel Xeon X5650
Clock	2.67 GHz
Hyperthreading	Disabled
Cores per processor chip	6
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	128 KiB
L3 cache/socket	12 MiB
DRAM	49 GiB
Compiler	Intel C/C++ compiler v13.1.1

Figure 4-7: Technical specifications of the machine used for benchmarking.

different scheduling methodologies. In particular, to better understand the performance properties of PRISM, we developed four scheduling algorithms for comparison:

- SERIAL-DDGC is an implementation of the serial scheduling algorithm from Figure 4-1. SERIAL-DDGC provides a serial performance baseline for measuring the parallel speedup achieved by the other, more complex, scheduling algorithms for dynamic data-graph computations.
- CILK+LOCKS is a lock-based scheduling algorithm for dynamic data-graph computations. During each round, CILK+LOCKS updates only an active subset of the vertices in the graph. It uses a locking scheme to avoid executing conflicting updates in parallel. The locking scheme associates a shared-exclusive (i.e., reader-writer) lock [103] with each vertex in the graph. Prior to executing an update $f(v)$, vertex v's lock is acquired exclusively, and a shared lock is acquired for each $u \in \text{Adj}[v]$. A global ordering of locks is used to avoid deadlock.
- RRLOCKS is the lock-based dynamic scheduling algorithm implemented by the round-robin ***sweep scheduler*** in the original shared-memory version of GraphLab. A bit vector *active* is used to represent the active set of vertices. During each round, RRLOCKS scans each vertex in the active set in a round-robin fashion, conditionally updating a vertex v_i if *active*$[i]$ is TRUE. To avoid races, a locking strategy is used to coordinate updates that conflict.
- RRCOLOR is a coloring-based dynamic scheduling algorithm that uses a bit vector *active* to represent the active set of vertices. Instead of using locks to coordinate conflicting updates, however, RRCOLOR uses a vertex-coloring of the graph. At the start of the computation, RRCOLOR partitions the vertices by color and stores them in static arrays. For a graph colored using χ colors, each round of the computation is divided into χ color steps. During the kth color step, RRCOLOR scans all color-k vertices and conditionally updates a color-k vertex v_i if *active*$[i]$ is TRUE.

Overheads for locking and for chromatic scheduling

We compared the overheads associated with coordinating conflicting updates of a dynamic data-graph computation using locks versus using chromatic scheduling. We evaluated these overheads by comparing the 12-core execution times for PRISM and CILK+LOCKS to execute the PageRank [68] data-graph computation on a suite of graphs. We used PageRank for this study because of its comparatively cheap update function, which makes overheads due to scheduling more pronounced. PageRank updates a vertex v by first scanning v's incoming edges to aggregate the data from its incoming neighbors, and then by scanning v's outgoing edges to activate its outgoing neighbors.

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

Figure 4-8: Performance of Prism versus Cilk+Locks when executing $10 \cdot |V|$ updates of the PageRank [68] data-graph computation on a suite of six real-world graphs and six synthetic graphs. Column "*Graph*" identifies the input graph, and columns $|V|$ and $|E|$ specify the number of vertices and edges in the graph, respectively. Column χ gives the number of colors Prism used to color the graph. Columns "Cilk+Locks," "Prism," and "Prism-R" present 12-core running times in seconds for each respective scheduler. Each running time is the median of 5 runs. Column "*Coloring*" gives the percentage of Prism's running time spent coloring the graph. Prism-R, discussed in Section 4.6, provides deterministic support for associative operations on global variables.

We executed the PageRank application on a suite of six synthetic and six real-world graphs. The six real-world graphs came from the Stanford Large Network Dataset Collection (SNAP) [250], and the University of Florida Sparse Matrix Collection [107]. The six synthetic graphs were generated using the "randLocal," "powerLaw," "gridGraph," and "rMatGraph" generators included in the Problem Based Benchmark Suite [360]. We chose the graphs in this suite to be large enough to stress the memory system and thus make parallel speedup comparatively difficult. That is, given the random access inherent in data-graph computations, we expect most references to vertex data to come from DRAM, making DRAM bandwidth a scarce shared commodity. Since the span of Prism is superconstant, however, for a fixed number of workers, increasing the size of the graph only increases parallelism, making good parallel speedup comparatively easy. Thus, we have pessimistically chosen the graphs in the suite to be large enough to make DRAM bandwidth a shared bottleneck but not unduly larger.

To isolate scheduling overheads, we forced Prism and Cilk+Locks to perform the same number of updates. We observed that Prism often performs slightly fewer rounds of updates than Cilk+Locks when both are allowed to run until convergence. We controlled this variation by explicitly setting the total number of updates on a graph to 10 times the number of vertices.

Figure 4-8 presents the empirical results for this study. The figure shows that over the 12 benchmark graphs, Prism executes between 1.0 and 2.1 times faster than Cilk+Locks on PageRank, exhibiting a geometric-mean speedup of a factor of 1.5. Moreover, the figure shows that an average of 10.9 % of Prism's total running time is spent coloring the data graph, which is approximately equal to the cost of executing $|V|$ updates. Prism colors the data-graph once to execute the data-graph computation, however, meaning that the cost of coloring can be amortized over all of the updates in the data-graph computation. By contrast, the locking scheme implemented by Cilk+Locks incurs overhead for every update. Before updating a vertex v, Cilk+Locks acquires the lock associated with v and

Benchmark	χ	*Updates*	RRLocks	RRColor	Prism	Prism-R
PR/L	333	48,475,700	35.3	14.5	17.7	18.4
ID/2000	4	40,000,000	63.2	50.1	59.2	59.9
FBP/C3	2	16,001,900	11.9	8.8	8.8	8.9
ID/1000	4	10,000,000	15.7	12.6	14.9	15.0
PR/G	43	9,164,280	3.1	1.3	2.1	2.2
FBP/C1	2	8,783,100	5.9	4.7	4.8	4.8
ALS/N	6	1,877,220	65.7	52.4	52.8	53.5

Figure 4-9: Performance of three schedulers on the seven application benchmarks from Figure 4-2, modified so that all vertices are activated in every round. Column "*Updates*" specifies the number of updates performed in the data-graph computation. Columns "RRLocks," "RRColor," "Prism," and "Prism-R" list the 12-core running times in seconds for the respective schedulers to execute each benchmark. Each running time is the median of 5 runs. The Prism-R algorithm, which provides deterministic support for associative operations on global variables, will be discussed in Section 4.6.

each lock associated with a vertex $u \in \text{Adj}[v]$. For simple data-graph computations whose update functions perform relatively little work, this step can account for a significant fraction of the time to execute an update.

Dynamic-scheduling overhead

To investigate the overhead of using multibags to maintain activation sets, we compared the 12-core running times of Prism, RRColor, and RRLocks on the seven benchmark applications from Figure 4-2. For this study, we modified the benchmarks slightly for each scheduler in order to provide a fair comparison. First, because Prism typically executes fewer updates than a scheduler for static data-graph computations, we modified the update functions for each application so that every update on a vertex v always activates all vertices $u \in \text{Adj}[v]$. This modification guarantees that Prism executes the same set of updates each round as RRLocks and RRColor, while still incurring the overhead that Prism requires in order to maintain a dynamic set of active vertices. Thus, we compare the worst case conditions for Prism with respect to scheduling overhead with the best case conditions for RRLocks and RRColor.

Figure 4-9 presents the results of these tests, revealing the overhead Prism incurs to maintain its activation sets using a multibag. As can be seen from the figure, Prism is 1.0 to 1.6 times slower than RRColor on the benchmarks with a geometric-mean relative slowdown of 1.16. That is, for static data-graph computations, Prism incurs only an aggregate 16 % slowdown through the use of a multibag, as opposed to the simple array used by RRColor, which suffices for static scheduling. The Prism algorithm, which can also support *dynamic* activation sets efficiently, incurred minimal overhead for the multibag data structure. Prism outperformed RRLocks on all benchmarks, achieving a geometric-mean speedup of 30 % due to RRLocks's lock overhead. Thus, Prism incurs relatively little overhead by maintaining activation sets with multibags.

The relative overhead of RRColor and Prism depends on the percentage of vertices active during a given round. As a typical example, RRColor is approximately 1.09 times faster than Prism on the image denoise benchmark when 80 % of the vertices are active each round, but is 1.11 times slower when 5 % or less of the vertices are active each round. As part of an effort to incorporate the Prism scheduling paradigm into an existing data-graph computation framework (e.g., GraphLab, Pregel etc.), one might consider using a heuristic to switch between the use of a bitvector and a multibag depending on the density of the

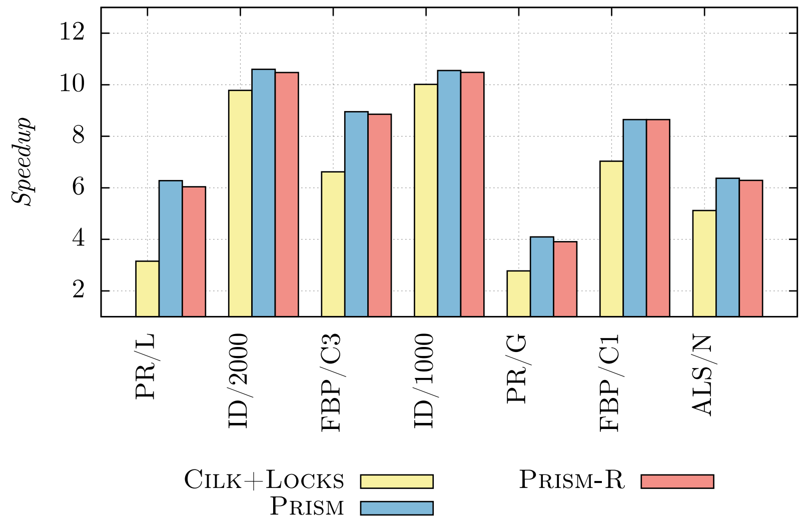


Figure 4-10: Empirical speedup relative to Serial-DDGC on 12 processor cores. Shown are the empirical speedups T_s/T_{12} of Cilk+Locks, Prism, and Prism-R, where T_s is the running time of the serial scheduling algorithm Serial-DDGC and T_{12} is the running time of the particular algorithm on 12 cores. The Prism-R algorithm is discussed in Section 4.6.

activation set. A simple heuristic such as a fixed threshold on the relative density of the activation set (e.g., 10 % of the vertices) would likely suffice to maintain activation sets with good performance: if fewer than 10 % of vertices are active, use a multibag, otherwise use a bitvector.

Scalability of Prism

To measure the scalability of Prism, and Cilk+Locks, we compared their 12-core running times to the running time of the serial reference implementation Serial-DDGC. Figure 4-10 shows the empirical 12-core speedups relative to Serial-DDGC of Prism and Cilk+Locks on seven application benchmarks. In geometric mean, Cilk+Locks achieved 5.73 times speedup and Prism achieved 7.56 times speedup.

In order to study the effect of the number χ of colors used to color the application's data graph on the parallel scalability of Prism, we measured the parallelism T_1/T_∞ and the 12-core speedup T_1/T_{12} of Prism while executing the image-denoise application as we varied the number of colors used. The image-denoise application performs belief propagation to remove Gaussian noise added to a gray-scale image. The data graph for the image-denoise application is a two-dimensional grid in which each vertex represents a pixel, and there is an edge between any two adjacent pixels. The Color-Graph procedure invoked in line 15 of Figure 4-3 typically colors this data-graph with just 4 colors.

To perform this study, we artificially increased χ by repeatedly assigning a new color to a random nonempty subset of the largest set of vertices with the same color. Using this technique, we ran the image-denoise application on a 500-by-500 pixel input image for values of χ between 4 and $250,000$, the last data point corresponding to a coloring that assigns all pixels distinct colors. Figure 4-11 plots the results of these tests. Although the parallelism of Prism is inversely proportional to χ, Prism's speedup on 12 cores is relatively insensitive to χ, as long as the parallelism is greater than about 120. This result is consistent with the

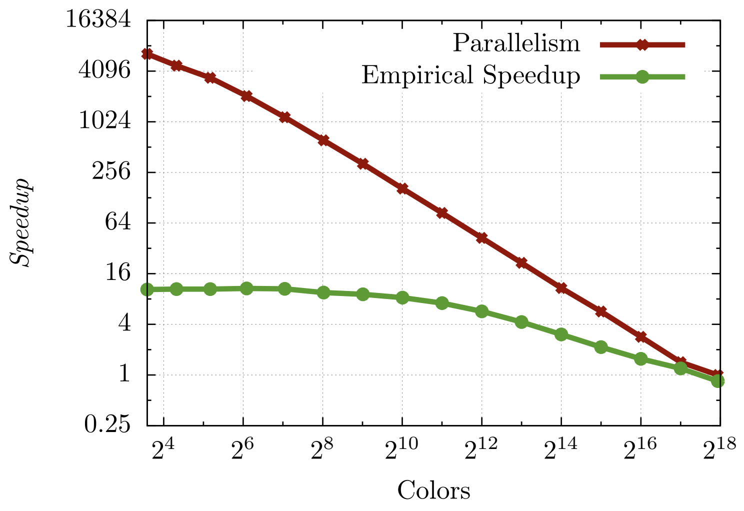


Figure 4-11: Scalability of PRISM on the image-denoise application as a function of χ, the number of colors used to color the data graph. The parallelism T_1/T_∞ is plotted together with the empirical speedup T_1/T_{12} achieved on a 12-core execution. Parallelism values were measured using the Cilkview scalability analyzer [180].

rule of thumb that a program with at least $10P$ parallelism should achieve nearly perfect linear speedup on P processors [100, p. 783].

4.6 The PRISM-R algorithm

This section introduces PRISM-R, a chromatic-scheduling algorithm that executes a dynamic data-graph computation deterministically even when updates modify global ***reducer variables*** using associative operations such as a reducer hyperobject [144]. While the chromatic scheduling technique employed by PRISM ensures that there are no data races on the vertex data of the graph, the order in which updates are made to a reducer variable among vertices of a common color can yield a nondeterministic result to the final reducer variable value. PRISM-R uses the "multivector" data structure, which is a theoretical improvement to the multibag, to maintain activation sets that are partitioned by color and ordered deterministically. We describe an extension of the model of simple data-graph computations that permits an update function to perform associative operations on global variables using a parallel reduction mechanism. In this extended model, PRISM-R executes dynamic data-graph computations deterministically while achieving the same work and span bounds as PRISM.

Data-graph computations with global reductions

Several frameworks for executing data-graph computations allow updates to modify global variables in limited ways. Pregel aggregators [269], and GraphLab's sync mechanism [265], for example, both support data-graph computations in which an update can modify a global variable in a restricted manner. These mechanisms coordinate parallel modifications to a global variable using parallel reductions. We shall focus our attention on allowing updates to modify global reducer hyperobjects, because reducers provide a particularly general re-

```
PRISM-R($G, f, Q_0$)
40  $\chi$ = COLOR-GRAPH($G$)
41  $r = 0$
42  $updates = 0$
43  $Q = Q_0$
44  while $Q \neq \emptyset$
45      $\mathcal{C}$ = MV-COLLECT($Q$)
46      for $C \in \mathcal{C}$
47          parallel for $i = 1, 2, \ldots, |C|$
48              $\langle v, p \rangle = C[i]$
49              if $p == priority[v]$
50                  $rank[f(v)] = updates + i$
51                  $priority[v] = \infty$
52                  $S = f(v)$
53                  parallel for $u \in S$
54                      if PRIORITYWRITE($priority[u], rank[f(v)]$)
55                          MV-INSERT($Q, \langle u, rank[f(v)] \rangle, color[u]$)
56          $updates = updates + |C|$
57      $r = r + 1$
```

```
PRIORITYWRITE($current, value$)
58  begin atomic
59  if $current > value$
60      $current = value$
61      return TRUE
62  else
63      return FALSE
64  end atomic
```

Figure 4-12: Pseudocode for PRISM-R. The algorithm takes as input a data graph G, an update function f, and an initial activation set Q_0. COLOR-GRAPH colors a given graph and returns the number of colors it used. The procedures MV-COLLECT and MV-INSERT operate the multivector Q to maintain activation sets for PRISM-R. PRISM-R updates the value of *updates* after processing each color set and r after each round of the data-graph computation.

duction mechanism that requires only associativity out of its binary reduction operator. Other parallel reduction mechanisms, including Pregel aggregators and GraphLab's sync mechanism, provide this guarantee only if the reduction operator is also commutative.

Although PRISM is implemented in Cilk Plus, PRISM does not produce a deterministic result if updates modify global variables using a noncommutative reducer. The reason for this is, in part, that the order of vertices within in a multibag depends on how the computation happens to be scheduled among participating workers. As a result, the order in which lines 21–26 of PRISM in Figure 4-3 evaluates the vertices in a color set C is nondeterministic. If two updates on vertices in C modify the same reducer, then the relative order of these modifications can differ between runs of PRISM, even if a single worker happens to execute both updates.

PRISM-R

PRISM-R is an extension to PRISM that executes dynamic data-graph computations deterministically even when update functions are allowed to perform associative operations on global variables. The semantics of PRISM-R mimic that of SERIAL-DDGC when its queue of active vertices is stable-sorted by color at the start of each round. In this modified version of SERIAL-DDGC updates to active vertices of the same color are applied in increasing order of their insertion into the queue. PRISM-R guarantees that the result of associative reductions performed by update functions reflect this same order.

Figure 4-12 presents the pseudocode for PRISM-R, which differs from PRISM in its use of a "multivector," instead of a multibag, to maintain partitioned activation sets, and in its use of a priority deduplication strategy for avoiding multiple updates to the same vertex in a round.

PRISM-R uses a ***multivector*** to represent a list of χ ***vectors*** (ordered multisets). A multivector supports the operations MV-INSERT and MV-COLLECT, which are analogous to the multibag operations MB-INSERT and MB-COLLECT, respectively. Each vector maintained by a multivector has serial semantics, meaning that the order of elements within each vector is deterministic and equivalent to the insertion order in an execution of the serial elision of the parallel program. Section 4.7 describes and analyzes the implementation of the multivector data structure.

The serial semantics of the multivector are not alone sufficient to ensure that updates are ordered deterministically in an execution of the serial elision of the program. Consider, for example, a round of PRISM that updates the three vertices x, y, z in parallel. Suppose that y activates u and both x and z activate a common neighbor v. The atomic compare-and-swap operator used by PRISM on line 25 of Figure 4-3 ensures that the updates on x and z will not both insert v into the activation set, but which of the two succeeds is nondeterministic. Inserting these two activated vertices into a multivector would produce either the order u, v or v, u depending on whether x or z activated v.

To eliminate this source of nondeterminism, PRISM-R assigns each update $f(v)$ a unique integer $\mathit{rank}[f(v)]$ on line 50 of Figure 4-12 that orders updates applied during a round according to their serial execution order of PRISM-R. Instead of maintaining a bit vector denoting whether or not a vertex is active, PRISM-R maintains an integer array *priority* of priorities. For each active vertex v, the value $\mathit{priority}[v]$ is equal to the smallest rank of any update $f(u)$ that activated v in the previous round. The priority of a vertex v is reset on line 51 before applying $f(v)$ by setting $\mathit{priority}[v] = \infty$.

For each vertex $u \in \mathrm{Adj}[v]$ activated by update $f(v)$, PRISM-R uses an atomic ***priority-write*** operator [359] to set $\mathit{priority}[u] = \min\{\mathit{priority}[u], \mathit{rank}[f(v)]\}$ and inserts the vertex-priority pair $\langle u, \mathit{rank}[f(v)]\rangle$ into the multivector if the priority write is successful on line 54. The color sets returned by MV-COLLECT on line 45 can contain multiple vertex-priority pairs for each active vertex. On lines 47–55 PRISM-R iterates over the vertex-priority pairs $\langle v, p\rangle$ in a color set and only applies the update $f(v)$ if $\mathit{priority}[v] == p$. Since $\mathit{priority}[v]$ is equal to the lowest ranked update that activated v, PRISM-R updates each active vertex exactly once during a round in the same order as a serial execution.

4.7 The multivector data structure

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

A ***multivector*** represents a list of χ ***vectors*** (ordered multisets). It supports the operations MV-INSERT and MV-COLLECT, which are analogous to the multibag operations MB-INSERT and MB-COLLECT, respectively.

Our implementation of multivector relies on how the runtime system in a dynamic multithreading concurrency platform executes a program with respect to the serial execution order of the program. Let $R(A)$ denote the sequence of strands executed in the serial execution of a program modeled by the dag A. The runtime system partitions $R(A)$ into a

```
FLATTEN($L, A, i$)
65  $A[i] = L$
66  if $L.left \neq$ NIL
67      spawn FLATTEN($L.left, A, i - L.right.size - 1$)
68  if $L.right \neq$ NIL
69      FLATTEN($L.right, A, i - 1$)
70  sync
```

Figure 4-13: Pseudocode for the FLATTEN operation for a log tree. FLATTEN performs a post-order parallel traversal of a log tree to place its nodes into a contiguous array.

```
IDENTITY()
71  $L$ = new log-tree node
72  $L.sublog$ = new vector
73  $L.size = 1$
74  $L.left$ = NIL
75  $L.right$ = NIL
76  return $L$
```

```
REDUCE($L_l, L_r$)
77  $L$ = IDENTITY()
78  $L.size = L_l.size + L_r.size + 1$
79  $L.left = L_l$
80  $L.right = L_r$
81  return $L$
```

Figure 4-14: Pseudocode for the IDENTITY and REDUCE log-tree reducer operations. The IDENTITY operation creates and returns a new log-tree node L. The REDUCE(L_l, L_r) operation concatenates a left log-tree node L_l with a right log-tree node L_r.

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

The log-tree reducer

A multivector stores its nonempty trace-local subvectors in a ***log tree***, which represents an ordered multiset of elements and supports $\Theta(1)$-work append operations. A log tree is a binary tree in which each node L stores a dynamic array $L.sublog$. The ordered multiset represented by a log tree corresponds to a concatenation of the tree's dynamic arrays in a post-order tree traversal. Each log-tree node L is augmented with the size of its subtree $L.size$ counting the number of log-tree nodes in the subtree rooted at L. Using this augmentation, the operation FLATTEN($L, A, L.size - 1$) described in Figure 4-13 flattens a log tree rooted at L of n nodes and height h into a contiguous array A using $\Theta(n)$ work and $\Theta(h)$ span.

To handle parallel MV-INSERT operations, a multivector employs a ***log-tree reducer***, that is, a reducer whose view type is a log tree. Figure 4-14 presents the pseudocode for the IDENTITY and REDUCE operations for the log-tree reducer.

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

```
A($R$)
82  Log-Insert($R, e_1$)
83  spawn B($R$)
84  Log-Insert($R, e_7$)
85  sync
86  Log-Insert($R, e_8$)

B($R$)
87  Log-Insert($R, e_2$)
88  spawn Log-Insert($R, e_3$)
89  Log-Insert($R, e_4$)
90  Log-Insert($R, e_5$)
91  sync
92  Log-Insert($R, e_6$)

Log-Insert($R, e$)
93  $L$ = Get-Local-View($R$)
94  Append($L.subblog, e$)
```

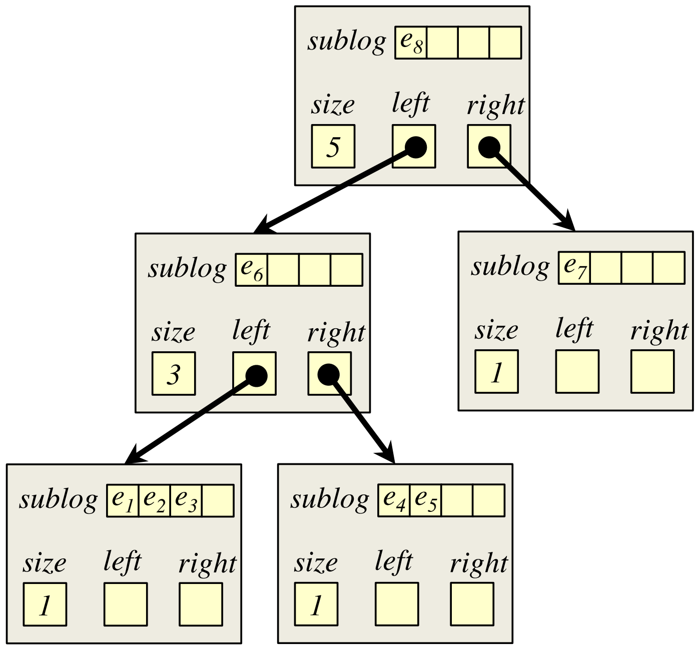


Figure 4-15: The state of a log-tree reducer R after a parallel execution of A(R). The runtime system creates new trace starting at line 84 of A and another starting at line 89 of B. As a result, the **sync** instructions in A and B therefore also create new traces, meaning that the serial execution order of this program is partitioned into 5 traces total. The ordered multiset $(e_1, e_2, \ldots, e_8)$ is represented by 5 trace-local sublogs ordered according to a post-order traversal of the log tree.

obtained by using a linked-list reducer in place of the log-tree reducer.

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

Figure 4-15 illustrates the state of a log-tree reducer R following the execution of the given fork-join parallel program A. The runtime system partitions the serial execution order of A at line 84 of A and line 89 of B. As a result, the **sync** instructions in A and B also partition the serial execution order, yielding a total of 5 traces. The log-tree reducer thus contains 5 nodes, one for each trace.

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

Implementation of MV-Insert *and* MV-Collect

Figure 4-16 sketches the MV-Insert(Q, v, k) operation to insert element v into the vector $C_k \in Q$. MV-Insert differs from MB-Insert in two ways. First, when a new subvector is created and added to a SPA, lines 100–101 additionally append that subvector to $Q.log\text{-}reducer$, thereby maintaining the log-tree reducer. Second, lines 96–97 reset the contents of the SPA $Q[p]$ after worker p begins executing a new trace, thereby ensuring that $Q[p]$ stores only trace-local subvectors.

```
MV-INSERT$(Q, v, k)$
 95  $p$ = GET-WORKER-ID()
 96  if worker $p$ began a new trace since last insert
 97        reset $Q[p]$
 98  if $Q[p].array[k]$ == NIL
 99        $Q[p].array[k]$ = new subvector
100        $L$ = GET-LOCAL-VIEW($Q.log\text{-}reducer$)
101        APPEND($L.sublog, Q[p].array[k]$)
102  APPEND($Q[p].array[k], v$)
```

Figure 4-16: Pseudocode for the MV-INSERT multivector operation. MV-INSERT(Q, v, k) inserts an element v into the kth vector C_k maintained by the multivector Q.

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

Figure 4-17: Pseudocode for the MV-COLLECT multivector operation. Calling MV-COLLECT on a multivector Q produces a pair $\langle$*vector-offsets*, *collected-subvectors*$\rangle$ of arrays, where *collected-subvectors* contains all nonempty subvectors in Q sorted by their associated vector's color, and *vector-offsets* associates sets of subvectors in Q with their corresponding vector.

Figure 4-17 sketches the MV-COLLECT operation, which, like its analog, MB-COLLECT, returns a pair $\langle$*subvector-offsets*, *collected-subvectors*$\rangle$. The procedure MV-COLLECT differs from MB-COLLECT primarily in that Step 1, which replaces Steps 1 and 2 in MB-COLLECT, flattens the log tree underlying $Q.log\text{-}reducer$ to produce the unsorted array *collected-subvectors*. MV-COLLECT also requires that *collected-subvectors* be sorted using a stable sort on Step 2. The integer sort described in the proof of Lemma 17 for MB-COLLECT is a suitable stable sort for this purpose.

Analysis of multivectors

We now analyze the work and span of the MV-INSERT and MV-COLLECT operations.

Lemma 22 *Executing* MV-INSERT *takes* $\Theta(1)$ *time in the worst case.*

PROOF. Resetting the SPA $Q[p]$ on line 97 can be done in $\Theta(1)$ worst-case time with an appropriate SPA implementation, and appending a new subvector to a log tree takes $\Theta(1)$ time. The lemma thus follows from the analysis of MB-INSERT in Lemma 16. □

Lemma 23 bounds the work and span of MV-COLLECT.

Lemma 23 *Let* Q *be a multivector of* χ *vectors whose contents are distributed across* m *subvectors. Then a call to* MV-COLLECT(Q) *in a dynamic-multithreaded computation with span* T_∞ *incurs* $\Theta(m + \chi)$ *work and* $\Theta(\lg m + \chi + T_\infty)$ *span.*

PROOF. Flattening the log-tree reducer in Step 1 is accomplished in two steps. First, the FLATTEN operation writes the nodes of the log tree to a contiguous array. Execution of FLATTEN has span proportional to the depth of the log tree, which is bounded by $O(T_\infty)$, since at most $O(T_\infty)$ reduction operations can occur along any path in A, and REDUCE for log trees executes in $\Theta(1)$ work [144]. Second, using a parallel-prefix sum computation, the log entries associated with each node in the log tree can be packed into a contiguous array, incurring $\Theta(m)$ work and $\Theta(\lg m)$ span. Step 1 thus incurs $\Theta(m)$ work and $O(\lg m + T_\infty)$ span. The remaining steps of MV-COLLECT, which are analogous to those of MB-COLLECT and analyzed in Lemma 17, execute in $\Theta(m + \chi)$ work and $\Theta(\chi + \lg m)$ span. □

4.8 Analysis and evaluation of PRISM-R

This section presents a theoretical work-span analysis of PRISM-R and an empirical analysis of PRISM-R. The theoretical analysis demonstrates that the work and span of PRISM-R are essentially the same as the work and span of PRISM. Empirically, PRISM-R is only 2%–7% slower than PRISM, overall, while providing deterministic support for associative operations on global variables.

Work-span analysis of PRISM-R

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

Theorem 24 *Let G be a Δ-degree data graph. Suppose that* PRISM-R *colors G using χ colors. Then* PRISM-R *executes updates on all vertices in the activation set Q_r for a round r of a simple data-graph computation $\langle G, f, Q_0\rangle$ in $\Theta(size(Q_r))$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta))$ span.*

PROOF. PRISM-R can perform a priority write to its *active* array with $\Theta(1)$ work, and it can remove duplicates from the output of MV-COLLECT in work $\Theta(size(Q_r))$ and span $O(\lg(size(Q_r))) = O(\lg Q_r + \lg \Delta)$. The theorem follows by applying Lemmas 22 and 23 appropriately to the analysis of PRISM in Theorem 19. □

Theorem 25 *Suppose that* PRISM-R *colors a Δ-degree data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0\rangle$ in r rounds applying updates to the activation sets $Q_0, Q_1, \ldots, Q_{r-1}$. Define the multiset $\mathcal{U} = \biguplus_{i=0}^{r-1} Q_i$ so that $|\mathcal{U}| = \sum_{i=0}^{r-1} |Q_i|$ and $size(\mathcal{U}) = \sum_{i=0}^{r-1} size(Q_i)$. Then* PRISM-R *executes the data-graph computation using $\Theta(size(\mathcal{U}))$ work and $O(r \cdot \chi(\lg((\mathcal{U}/r)/\chi) + \lg \Delta))$ span.*

PROOF. By Theorem 24 PRISM-R executes a round of a data-graph computation using the similar asymptotic work and span as PRISM. We mirror the arguments in Theorem 20 to bound the work and span of PRISM-R for a sequence of rounds. □

Given Theorem 25 we can compute the parallelism of PRISM-R for a data-graph computation that applies a multiset $\mathcal{U}$ of updates over r rounds. The following corollary expresses the parallelism of PRISM-R in terms of the average size of the activation sets in a sequence of rounds.

Corollary 26 *Suppose* PRISM-R *executes a data-graph computation in r rounds during which it applies a multiset $\mathcal{U}$ of updates. Define the average number of updates per round*

$U_{avg} = |\mathcal{U}|/r$ and the average work per round $W_{avg} = size(\mathcal{U})/r$. Then PRISM-R *has $\Omega(W_{avg}/(\chi(\lg(U_{avg}/\chi) + \lg\Delta)))$ parallelism.*

PROOF. The theorem follows from Theorem 25 by computing the parallelism as the ratio of the work over the span and then performing substitution. □

***Empirical evaluation of* PRISM-R**

PRISM-R provides deterministic support for associative operations on global variables at the cost of additional complexity versus PRISM, specifically in the maintenance of activation sets. Nonetheless, PRISM-R guarantees the essentially the same asymptotic work and span as PRISM. Empirically, we find that PRISM-R suffers a geometric-mean slowdown of only 2 %–7 % versus PRISM in various scenarios. In particular, the 12-core performance for each dynamic data-graph computation application featured in Figure 4-2 demonstrate that for real-world applications PRISM-R is 7 % slower in geometric mean than PRISM. In Figure 4-9 we see that PRISM-R is only 1.8 % slower than PRISM for static versions of the applications featured in Figure 4-2 (i.e., all vertices are updated every round). In Figure 4-8 we present the 12-core performance of PRISM-R on PageRank [68] for a suite of six synthetic and six real-world graphs. PRISM-R in this case is 3.5 % slower in geometric mean than PRISM. Finally, Figure 4-10 shows that PRISM-R achieved 7.42 times speedup on 12 cores compared to the serial reference implementation SERIAL-DDGC, which is only 1.9 % less speedup than PRISM.

4.9 Conclusion

The behavior of PRISM corresponds to a variant of SERIAL-DDGC that sorts the activated vertices in its queue by color at the start of each round. Whether PRISM executes a given data graph on 1 processor or many, it always behaves the same way. With PRISM-R, this property holds even when the update function can perform reductions (e.g., associative operators on global variables). By contrast, lock-based schedulers provide no such a guarantee of determinism. Instead, updates in a round executed by a lock-based scheduler appear to execute according to some linear order, the so-called *sequential consistency* model employed by GraphLab [264, 265] and others. This order is nondeterministic due to races on the acquisition of locks. PRISM and PRISM-R thus provide a deterministic solution to executing dynamic data-graph computations efficiently, which allows programmers to reason about their behavior in a principled manner.

Blelloch *et al.* [52] argue that deterministic programs can be fast compared with nondeterministic programs, and they document many examples where the overhead for converting a nondeterministic program into a deterministic one is small. They even document a few cases where this "price of determinism" is *slightly* negative. To their list, we add the execution of dynamic data-graph computations as having a price of determinism which is *significantly* negative.

Chapter 5

Ordering Heuristics for Parallel Graph Coloring

This chapter examines parallel algorithms for efficiently coloring graphs and presents ordering heuristics for greedy graph-coloring algorithms [175]. This work was conducted in collaboration with William Hasenplaugh, Tim Kaler, and Charles E. Leiserson.

5.1 Introduction

Graph coloring is a heavily studied problem with many real-world applications, including the scheduling of conflicting jobs [22, 136, 272, 403], register allocation [67, 83, 84], high-dimensional nearest-neighbor search [40], and sparse-matrix computation [94, 210, 337], to name just a few. Formally, a ***(vertex)-coloring*** of an (undirected) graph $G = (V, E)$ is an assignment of a ***color*** $v.\mathit{color}$ to each vertex $v \in V$ such that for every edge $(u, v) \in E$, we have $u.\mathit{color} \neq v.\mathit{color}$, that is, no two adjacent vertices have the same color. The ***graph-coloring problem*** is the problem of determining a coloring which uses as few colors as possible.

We were motivated to work on graph coloring in the context of the work presented in Chapter 4 on chromatic scheduling of parallel data-graph computations. Let us briefly review chromatic scheduling and data-graph computations here. A data graph is a graph with data associated with its vertices and edges. A data-graph computation is an algorithm implemented as a sequence of updates on the vertices of a data graph $G = (V, E)$, where updating a vertex $v \in V$ involves computing a new value associated with v as a function of v's old value and the values associated with the ***neighbors*** of v: the set of vertices adjacent to v in G, denoted $\mathrm{Adj}[v] = \{u \in V : (v, u) \in E\}$. To ensure atomicity of each update, rather than using mutual-exclusion locks or other nondeterministic means of data synchronization, chromatic scheduling first colors the vertices of G and then sequences through the colors, scheduling all vertices of the same color in parallel. The time to perform a data-graph computation thus depends both on how long it takes to color G and on the number of colors produced by the graph-coloring algorithm: more colors means less parallelism. Although the coloring can be performed offline for some data-graph computations, for other computations the coloring must be produced online, and one must accept a trade-off between coloring ***quality*** — number of colors — and the time to produce the coloring.

Although the problem of finding an ***optimal*** coloring of a graph — a coloring using the fewest colors possible — is in NP-complete [150], heuristic "greedy" algorithms work reasonably well in practice. Welsh and Powell [403] introduced the original ***greedy*** coloring algorithm, which iterates over the vertices and assigns each vertex the smallest color not assigned to a neighbor. For a graph $G = (V, E)$, define the ***degree*** of a vertex $v \in V$ by $\deg(v) = |\text{Adj}[v]|$, the number of neighbors of v, and let the ***degree*** of G be $\Delta = \max_{v \in V} \{\deg(v)\}$. Welsh and Powell show that the greedy algorithm colors a graph G with degree Δ using at most $\Delta + 1$ colors.

Ordering heuristics

In practice, however, greedy coloring algorithms tend to produce much better colorings than the $\Delta + 1$ bound implies, and moreover, the order in which a greedy coloring algorithm colors the vertices affects the quality of the coloring.[1] To reduce the number of colors a greedy coloring algorithm uses, practitioners therefore employ ***ordering heuristics*** to determine the order in which the algorithm colors the vertices [12, 65, 209, 275].

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

FF The ***first-fit*** ordering heuristic [263, 403] colors vertices in the order they appear in the input graph representation.

R The ***random*** ordering heuristic [209] colors vertices in a uniformly random order.

LF The ***largest-degree-first*** ordering heuristic [403] colors vertices in order of decreasing degree.

ID The ***incidence-degree*** ordering heuristic [94] iteratively colors an uncolored vertex with the largest number of colored neighbors.

SL The ***smallest-degree-last*** ordering heuristic [12, 275] colors the vertices in the order induced by first removing all the lowest-degree vertices from the graph, then recursively coloring the resulting graph, and finally coloring the removed vertices.

SD The ***saturation-degree*** ordering heuristic [65] iteratively colors an uncolored vertex whose colored neighbors use the largest number of distinct colors.

The experimental results overviewed in Section 5.8 indicate that we have listed these heuristics in rough order of coloring quality from worst to best, confirming the findings of Gebremedhin and Manne [153], who also rank the relative quality of R, LF, ID, and SD in this order.

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

Figure 5-1 gives the pseudocode for GREEDY, a greedy coloring algorithm. GREEDY takes a vertex-weighted graph $G = (V, E, \rho)$ as input, where $\rho : V \to \mathbb{R}$ is a priority function produced by some ordering heuristic. Each step of GREEDY simply selects the uncolored

[1]In fact, for any graph $G = (V, E)$, some ordering of V causes a greedy algorithm to color G optimally, although finding such an ordering is NP-hard [285].

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

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

Figure 5-1: Pseudocode for a serial greedy graph-coloring algorithm. Given a vertex-weighted graph $G = (V, E, \rho)$, where the priority of a vertex $v \in V$ is given by $\rho(v)$, GREEDY colors each vertex $v \in V$ in decreasing order according to $\rho(v)$.

```
JP($G$)
 7  let $G = (V, E, \rho)$
 8  parallel for $v \in V$
 9      $v.\mathit{pred} = \{u \in V : (u, v) \in E \text{ and } \rho(u) > \rho(v)\}$
10      $v.\mathit{succ} = \{u \in V : (u, v) \in E \text{ and } \rho(u) < \rho(v)\}$
11      $v.\mathit{counter} = |v.\mathit{pred}|$
12  parallel for $v \in V$
13      if $v.\mathit{pred} == \emptyset$
14          JP-COLOR($v$)
```

```
JP-COLOR($v$)
15  $v.\mathit{color}$ = GET-COLOR($v$)
16  parallel for $u \in v.\mathit{succ}$
17      if JOIN($u.\mathit{counter}$) == 0
18          JP-COLOR($u$)
```

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

Figure 5-2: The Jones-Plassmann parallel coloring algorithm. JP uses a recursive helper function JP-COLOR to process a vertex once all of its predecessors have been colored. JP-COLOR uses the helper routine GET-COLOR to find the smallest color available to color a vertex v.

vertex with the highest priority according to ρ and colors it with the smallest available color. Generally, for a coloring algorithm A and ordering heuristic H, let A-H denote the coloring algorithm A that runs on vertex-weighted graphs whose priority functions are produced by H. In this way, we separate the behavior of the coloring algorithm from that of the ordering heuristic.

GREEDY, using any of these six ordering heuristics, can be made to run in $\Theta(V + E)$ time theoretically. Although some of these ordering heuristics involve more bookkeeping than others, achieving these theoretical bounds for GREEDY-FF, GREEDY-R, GREEDY-LF, GREEDY-ID, and GREEDY-SL is straightforward [212, 275]. Despite conjectures to the contrary [94, 212], GREEDY-SD can also be made to run in $\Theta(V + E)$ time, as we shall show in Section 5.8.

In practice, producing a better quality coloring tends to cost more in running time. That is, the six heuristics, which are listed in increasing order of coloring quality, are also listed in increasing order of running time. The only exception is GREEDY-ID, which is dominated by GREEDY-SL in both coloring quality and running time. The experiments discussed in the Section 5.8 summarize our empirical findings for serial greedy coloring.

Parallel greedy coloring

A historical tension exists between coloring quality and the parallel scalability of greedy graph coloring. Although the traditional ordering heuristics FF, LF, ID, and SL are efficient using GREEDY, it can be shown that any parallelization of them requires worst-case span of $\Omega(V)$ for a general graph $G = (V, E)$. Of the various attempts to parallelize greedy coloring [93, 117, 266], the algorithm first proposed by Jones and Plassmann [209] extends the greedy algorithm in a straightforward manner, uses work linear in size of the graph, and is deterministic given any particular choice of random seed. Jones and Plassmann's original paper demonstrates good parallel performance for $O(1)$-degree graphs using the random ordering heuristic R. Unfortunately, in practice, R tends to produce colorings of relatively poor quality compared to the other traditional ordering heuristics. But the other traditional ordering heuristics are all vulnerable to adversarial graph inputs which cause JP to operate in $\Omega(V)$ time and thus exhibit poor parallel scalability. This state of affairs forces performance engineers working with graph coloring codes to choose between good coloring quality and parallel scalability. Consequently, there is need for new ordering heuristics for JP that can achieve both good coloring quality and guarantee parallel scalability.

Figure 5-2 gives the pseudocode for JP, which colors a given graph $G = (V, E, \rho)$ in the order specified by the priority function ρ. The algorithm begins on lines 9 and 10 by partitioning the neighbors of each vertex into ***predecessors*** — vertices with larger priorities — and ***successors*** — vertices with smaller priorities. JP uses the recursive JP-COLOR helper function to color a vertex $v \in V$ once all of v's predecessors, $v.pred$, have been colored. Initially, lines 12–14 in JP scan the vertices of V to find every vertex that has no predecessors and colors each one using JP-COLOR. Within a call to JP-COLOR(v), line 15 calls GET-COLOR to assign a color to v, and the loop on lines 16–18 broadcasts in parallel to all of v's successors, $v.succ$, the fact that v is colored. For each successor $u \in v.succ$, line 17 tests whether all of u's predecessors have already been colored, and if so, line 18 recursively calls JP-COLOR on u.

Jones and Plassmann analyze the performance of JP-R for $O(1)$-degree graphs. Although they do not discuss using the naive FF ordering heuristic, it is apparent that there exist adversarial input orderings for which their algorithm would fail to scale. For example, if the graph $G = (V, E)$ is simply a chain of vertices and the input order of V corresponds to their order in the chain, JP-FF exhibits no parallelism. Jones and Plassmann show that a random ordering produced by R, however, allows the algorithm to run in $O(\lg V / \lg\lg V)$ expected time on this chain graph and, for that matter, on any $O(1)$-degree graph. Section 5.3 of this chapter extends their analysis of JP-R to arbitrary-degree graphs.

Although JP-R scales well both in theory and in practice, when it comes to coloring quality, R is one of the weaker ordering heuristics, as we have noted. Of the other heuristics, JP-LF and JP-SL suffer from the same problem as FF, namely, it is possible to construct adversarial graphs that cause them to scale poorly, which we explore in Section 5.4. The ID heuristic tends to produce worse colorings than SL, and since GREEDY-ID also runs more slowly than GREEDY-SL, we have dropped ID from consideration. Moreover, because of our motivation to use the coloring algorithm for online chromatic scheduling, where the performance of the coloring algorithm cannot be sacrificed for marginal improvements in the quality of coloring, we also have dropped the SD heuristic. Because SD produces the best-quality colorings of the six ordering heuristics, however, we see parallelizing it as an interesting opportunity for future research.

Consequently, this chapter focuses on alternatives to the LF and SL ordering heuristics that provide comparable coloring quality while exhibiting the same resilience to adversarial graphs that R shows compared with FF. Specifically, we introduce two new randomized ordering heuristics — "largest log-degree first" (LLF) and "smallest log-degree last" (SLL)

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

Figure 5-3: Summary of ordering-heuristic behavior on a suite of 8 real-world graphs and 10 synthetic graphs when run on a machine with 12 Intel Xeon X5650 processor cores. Column H lists three serial heuristics traditionally used for Greedy, and column H' lists parallel heuristics for JP, of which LLF and SLL are introduced in this chapter. Column "$C_{H'}/C_H$" shows the geometric mean of the ratio of the number of colors the parallel heuristic uses compared to the serial heuristic. Column "Greedy-H/JP-H'_1" shows the geometric mean of the ratio of serial running times of Greedy with the serial heuristic versus JP with the analogous parallel heuristic when run on 1 processor. Column "JP-H'_1/JP-H'_{12}" shows the geometric mean of the speedup of each parallel heuristic going from 1 processor to 12.

— which resemble LF and SL, respectively, but which scale provably well when used with JP. We demonstrate that JP-LLF and JP-SLL provide good parallel scalability in theory and in practice and are resilient to adversarial graphs. By using the LLF and SLL heuristics, performance engineers can enjoy similar improvements to coloring quality as those that LF and SL bestow to serial graph-coloring codes, without sacrificing parallel performance.

Figure 5-3 summarizes our empirical findings. The data suggest that the LLF and SLL ordering heuristics produce colorings that are nearly as good as LF and SL, respectively. With respect to performance, our implementations of JP-LLF and JP-SLL actually operate slightly faster on 1 processor than our highly tuned implementations of Greedy-LF and Greedy-SL, respectively, and they scale comparably to JP-R.

The theoretical and empirical performance analyses presented in this chapter allow programmers to apply their predictions about the performance of parallel greedy graph-coloring codes to other contexts in which these codes are used. In particular, the analyses in this chapter complement the theoretical analyses of the Prism and Prism-R algorithms, introduced in Chapter 4, to fully justify that these algorithms scale well in parallel, even when the cost of coloring is taken into account.

Outline

The remainder of this chapter is organized as follows. Section 5.2 reviews the asynchronous parallel greedy coloring algorithm first proposed by Jones and Plassmann [209]. We show how JP can be extended to handle arbitrary-degree graphs and arbitrary priority functions. Using work-span analysis [100, Ch. 27], we show that JP colors a Δ-degree graph $G = (V, E, \rho)$ in $\Theta(V + E)$ work and $O(L \lg \Delta + \lg V)$ span, where L is the length of the longest path in G along which the priority function ρ decreases. Section 5.3 analyzes the performance of JP-R, showing that it operates using linear work and $O(\lg V + \lg \Delta \cdot \min\{\sqrt{E}, \Delta + \lg \Delta \lg V / \lg \lg V\})$ span. Section 5.4 shows that there exist "adversarial" graphs for which JP-LF and JP-SL exhibit limited parallel speedup. Section 5.5 introduces and analyzes the LLF and SLL ordering heuristics. We show that, given a Δ-degree graph G, JP-LLF colors $G = (V, E, \rho)$ using $\Theta(V + E)$ work and $O(\lg V + \lg \Delta(\min\{\Delta, \sqrt{E}\} + \lg^2 \Delta \lg V / \lg \lg V))$ expected span, while JP-SLL colors $G = (V, E, \rho)$ using same work and an additive $\Theta(\lg \Delta \lg V)$ additional span. Section 5.6 evaluates the performance of JP-LLF and JP-SLL on a suite of 8 real-world and 10 synthetic benchmark graphs. Section 5.7 discusses the software engineering techniques used in our implementation of JP-R, JP-LLF, and JP-SLL. Section 5.8 presents some experimental results for serial ordering heuristics and introduces an algorithm for computing the SD ordering heuristic using $\Theta(V+E)$ work. Section 5.9 discusses related work, and Section 5.10 offers some concluding remarks.

5.2 The Jones-Plassmann algorithm

This section reviews JP, the parallel greedy coloring algorithm introduced by Jones and Plassmann [209], whose pseudocode is given in Figure 5-2. We describe how JP can be modified from Jones and Plassmann's original algorithm to handle arbitrary-degree graphs and arbitrary priority functions. We analyze JP with an arbitrary priority function ρ and show that on a Δ-degree graph $G = (V, E, \rho)$, JP runs in $\Theta(V+E)$ work and $O(L \lg \Delta + \lg V)$ span, where L is the longest path in the "priority dag" of G induced by ρ.

The theoretical analysis throughout this chapter assume that the parallel computer supports read-modify-write instructions [183] and incurs no overhead due to contention.

Analysis of JP

To analyze the performance of JP, we shall think of JP as coloring the vertices in the partial order of a "priority dag," similar to the priority dag described by Blelloch *et al.* [53]. Specifically, on a vertex-weighted graph $G = (V, E, \rho)$, the priority function ρ induces a ***priority dag*** $G_\rho = (V, E_\rho)$, where $E_\rho = \{(u, v) \in V \times V : (u, v) \in E \text{ and } \rho(u) > \rho(v)\}$, that is, E_ρ contains a directed edge from u to v, where $\rho(u) > \rho(v)$, for each (undirected) edge $(u, v) \in E$. Because ρ is an injective function, it induces a total order on the vertices, and thus G_ρ is indeed a dag. We shall bound the span of JP running on a graph G in terms of the ***depth*** of G_ρ, that is, the length of a longest path through G_ρ.

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

Lemma 27 *The helper routine* GET-COLOR*, shown in Figure 5-2, can be implemented so that during the execution of* JP *on a graph* $G = (V, E, \rho)$*, a call to* GET-COLOR(v) *for a vertex* $v \in V$ *costs* $\Theta(k)$ *work and* $\Theta(\lg k)$ *span, where* $k = |v.pred|$.

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

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

Theorem 28 *Given a* Δ*-degree graph* $G = (V, E, \rho)$ *for some priority function* ρ*, let* G_ρ *be the priority dag induced on* G *by* ρ*, and let* L *be the depth of* G_ρ*. Then* JP(G) *runs in* $\Theta(V+E)$ *work and* $O(L \lg \Delta + \lg V)$ *span.*

PROOF. Let us first bound the work and span of JP-COLOR excluding any recursive calls. For a single call to JP-COLOR on a vertex $v \in V$, Lemma 27 shows that line 15 takes $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span. The JOIN operation on line 17 can be implemented as an atomic decrement-and-fetch operation [183] on the specified counter. Hence, excluding the recursive call, the loop on lines 16–18 performs $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span to decrement the counters of all successors of v.

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

5.3 JP with random ordering

This section bounds the depth of a priority dag G_ρ induced on a Δ-degree graph $G = (V, E, \rho)$ by a random priority function ρ in R. We show that the expected depth of G_ρ is $O(\min\{\sqrt{E}, \Delta + \lg \Delta \lg V / \lg\lg V\})$,[3] extending Jones and Plassmann's $O(\lg V/\lg\lg V)$ bound for the depth of G_ρ when $\Delta = \Theta(1)$ [209]. Combined with Theorem 28, this bound implies that JP-R executes in $O(\lg V + \lg \Delta \cdot \min\{\sqrt{E}, \Delta + \lg \Delta \lg V/\lg\lg V\})$ expected span.

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

Lemma 29 *The number of k-length simple paths in any Δ-degree graph $G = (V, E)$ is at most $|V| \cdot \min\{\Delta^{k-1}, (2|E|/(k-1))^{k-1}\}$.*

Proof. Consider selecting a k-length simple path $p = \langle v_1, \ldots, v_k \rangle$ in G. There are $|V|$ choices for v_1, and for all $i \in 1, 2, \ldots, k-1$, given a choice of $\langle v_1, \ldots, v_i \rangle$, there are at most $\deg(v_i)$ choices for v_{i+1}. Hence there are at most $J = |V| \cdot \prod_{i=1}^{k-1} \deg(v_i)$ simple paths in G of length k. Let $V_{k-1} \subseteq V$ denote some set of $k-1$ vertices in V, and let $\delta = \max_{V_{k-1}}\{\sum_{v \in V_{k-1}} \deg(v)/(k-1)\}$ be the maximum average degree of any such set. Then we have $J \le |V| \cdot \delta^{k-1}$.

The proof follows from two upper bounds on δ. First, because $\deg(v) \le \Delta$ for all $v \in V$, we have $\delta \le \Delta$. Second, for all $V_{k-1} \subseteq V$, we have $\sum_{v \in V_{k-1}} \deg(v) \le \sum_{v \in V} \deg(v) = 2|E|$ by the handshaking lemma [100, p. 1172–3], and thus $\delta \le 2|E|/(k-1)$. $\square$

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

Lemma 30 *Define the function $g(\alpha, \beta)$ for $\alpha, \beta > 1$ as*

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

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

Proof. We consider the cases when $\alpha \ge e^2$ and when $\alpha < e^2$ separately.

[3]Hasenplaugh improved this analysis to show that the expected depth of G_ρ is $\Theta(\min\{\sqrt{E}, \Delta\})$.

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

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

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

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

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

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

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

□

The following theorem applies Lemmas 29 and 30 to bound the depth of G_ρ.

Theorem 31 *Let $G = (V, E)$ be a Δ-degree graph, let $n = |V|$ and $m = |E|$, and let G_ρ be a priority dag induced on G by a random priority function $\rho \in \mathrm{R}$. For any constant $\epsilon > 0$ and sufficiently large n, with probability at most $n^{-\epsilon}$, there exists a directed path of length $e^2 \cdot \min\{\Delta, \sqrt{m}\} + (1+\epsilon)\min\{e^2 \ln \Delta \ln n / \ln \ln n, \ln n\}$ in G_ρ.*

Proof. Let $p = \langle v_1, \ldots, v_k \rangle$ be a k-length simple path in G. Because ρ is a random priority function, ρ induces each possible permutation among $\{v_1, \ldots, v_k\}$ with equal probability. If p is a directed path in G_ρ, then we must have that $\rho(v_1) < \rho(v_2) < \cdots < \rho(v_k)$. Hence, p is a k-length path in G_ρ with probability at most $1/k!$. If J is the number of k-length simple paths in G, then by the union bound, the probability that a k-length directed path exists in G_ρ is at most $J/k!$, which is at most $J(e/k)^k$ by Stirling's approximation [100, p. 57].

We consider cases when $\Delta < \ln n$ and $\Delta \geq \ln n$ separately.

First, suppose that $\Delta < \ln n$. By Lemma 29, the number of k-length simple paths in G is at most $J = n\Delta^{k-1} \leq n\Delta^k$, which implies that the probability that a k-length path exists in G_ρ is at most $n(e\Delta/k)^k$. We assume, without loss of generality, that $\Delta > 2$, because the theorem holds for $O(1)$-degree graphs as a result of [209]. For $\Delta \geq 2$, let $\alpha = \Delta$ and $\beta = \ln n$. Because $\alpha \geq 2$ and $\beta \geq e^2$ for sufficiently large n, Lemma 30 implies that the function $g(\alpha, \beta) = e^2(\ln \alpha / \ln \beta) \ln(\beta \ln \alpha / \alpha \ln \beta)$ is at least 1. Letting

$k = e^2(\Delta + (1+\epsilon)\ln\Delta \ln n / \ln\ln n)$, we conclude that

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

Now suppose that $\Delta \ge \ln n$. We consider the cases when $\Delta < \sqrt{m}$ and $\Delta \ge \sqrt{m}$, separately. When $\Delta < \sqrt{m}$, letting $k = e^2\Delta + (1+\epsilon)\ln n$, the theorem follows from the facts that $k \ge (1+\epsilon)\ln n$ and $k \ge e^2\Delta$. When $\Delta \ge \sqrt{m}$, let $k = e^2\sqrt{m} + (1+\epsilon)\ln n$. By Lemma 29, the number of k-length simple paths is at most $n(2m/(k-1))^{k-1} \le n(4m/k)^k$, and thus the probability that a k-length path exists in G_ρ is at most $n(4em/k^2)^k$. The theorem follows from the facts that $k \ge (1+\epsilon)\ln n$ and $k^2 \ge e^4 m$. □

Corollary 32 *Given a graph $G = (V, E, \rho)$, where $\rho \in \mathrm{R}$ is a random priority function, the expected depth of the priority dag G_ρ is $O(\min\{\sqrt{E}, \Delta + \lg\Delta \lg V / \lg\lg V\})$, and thus* JP-R *colors all vertices of G with $O(\lg V + \lg\Delta \cdot \min\{\sqrt{E}, \Delta + \lg\Delta \lg V/\lg\lg V\})$ expected span.*

PROOF. Theorems 28 and 31 together imply the corollary. □

5.4 The LF and SL heuristics

This section shows that the largest-first (LF) and smallest-last (SL) ordering heuristics can inhibit parallel speedup when used by JP. We examine a "clique-chain" graph and show that JP-LF incurs $\Omega(\Delta^2)$ span to color a Δ-degree "clique-chain" graph G, whereas JP-R colors G incurring only $O(\Delta \lg \Delta)$ expected span. We formally review the SL ordering heuristic and observe that this formulation of SL means that JP-SL requires $\Omega(V)$ span to color a path graph $G = (V, E)$, which JP-R colors in $O(\lg V)$ span. We shall see in Section 5.5 that it is possible to achieve coloring quality comparable to LF and SL, but with guaranteed parallel scalability comparable to JP-R.

The LF ordering heuristic

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

Although LF has been used in parallel greedy graph-coloring algorithms in the past [12, 212], Figure 5-4 illustrates a Δ-degree "clique-chain" graph $G = (V, E)$ for which JP-LF incurs $\Omega(\Delta^2)$ span to color, but JP-R colors with only $O(\Delta\lg\Delta + \lg^2\Delta \lg V/\lg\lg V)$ expected span. Conceptually, the ***clique-chain*** graph comprises a set of cliques of increasing size that are connected in a "chain" such that JP-LF is forced to color these cliques sequentially from largest to smallest. Figure 5-4 specifically illustrates a Δ-degree clique-chain graph, where 3 evenly divides Δ. This clique-chain graph contains a sequence of cliques $\mathcal{K} = \{K_1, K_4, \ldots, K_{\Delta-2}\}$ of increasing size, each pair of which is separated by two additional vertices forming a linear chain. Specifically, for $r \in \{1, 4, \ldots, \Delta - 2\}$, each vertex $u \in$

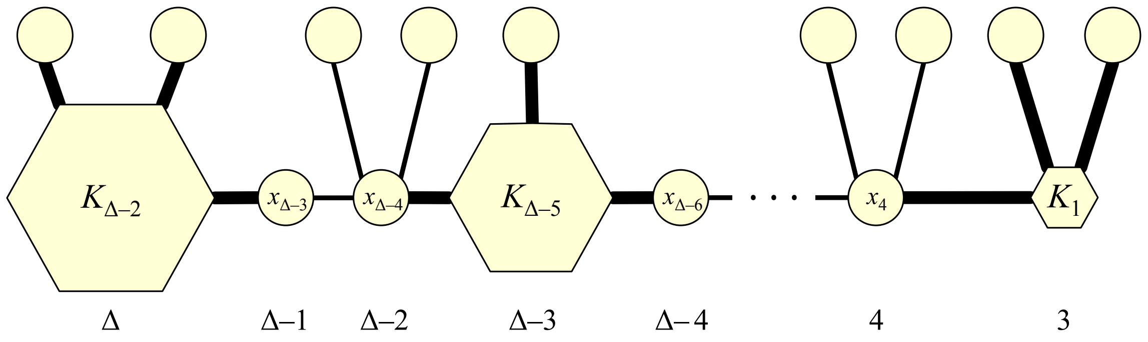


Figure 5-4: A Δ-degree clique-chain graph G, which Theorem 33 shows is adversarial for JP-LF. This graph contains $\Theta(\Delta^2)$ vertices arranged as a chain of cliques. Each hexagon labeled K_r represents a clique of r vertices, and circles represent individual vertices. A thick edge between an individual vertex and a clique indicates that the vertex is connected to every vertex within the clique. A label below an individual vertex indicates the degree of the associated vertex, and a label below a clique indicates the degree of every vertex within that clique.

K_r is connected to each vertex $u \in K_{r+3}$ by a path $\langle u, x_{r+1}, x_{r+2}, v\rangle$ for distinct vertices $x_{r+1}, x_{r+2} \in V$. Additional vertices, shown above the chain in Figure 5-4, ensure that the degree of each vertex in K_r is $r+2$, and the degrees of the vertices x_{r+1} and x_{r+2} are $r+3$ and $r+4$, respectively. Clique-chain graphs of other degrees are structured similarly.

The following theorem uses the clique chain graph to show that JP-LF can incur a large span to color a graph.

Theorem 33 *For any $\Delta > 0$, there exists a Δ-degree graph $G = (V, E)$ such that* JP-LF *colors G in $\Omega(\Delta^2)$ span and* JP-R *colors G in $O(\Delta \lg \Delta)$ expected span.*

Proof. Assume without loss of generality that 3 divides Δ and that G is a clique-chain graph. The span of JP-R follows from Corollary 32, noting that $|V| = \Theta(\Delta^2)$. Because JP-LF trivially requires $\Omega(1)$ span to process each vertex in G, we bound the span of JP-LF on G by showing that the length of the longest path p in the priority dag G_ρ induced on G by any priority function ρ in LF is $\Delta^2/6 + \Delta/2 + 2$. Because LF assigns higher priority to higher-degree vertices, p starts at some vertex in $K_{\Delta-2}$, which has degree Δ, and passes through the $\Delta - 2$ vertices in $K_{\Delta-2}$ followed by $x_{\Delta-3}$ and $x_{\Delta-4}$.[4] The remainder of p is a longest path through the clique-chain graph G' of degree $\Delta - 3$ in the remaining graph $G - K_{\Delta-2} - \{x_{\Delta-3}, x_{\Delta-4}\}$, which has a longest path p' of length $|p'| = (\Delta-3)^2/6 + (\Delta - 3)/2 + 2$ by induction. The length of p is thus $\Delta + |p'| = \Delta^2/6 + \Delta/2 + 2$. □

The SL ordering heuristic

We focus on the formulation of the SL ordering heuristic due to Allwright *et al.* [12], because our experiments indicate that it gives colorings using fewer colors than other formulations [275].

Given a graph $G = (V, E)$, the SL ordering heuristic produces a priority function ρ via an iterative algorithm that assigns priorities to the vertices V in rounds to induce an ordering on V. For $i > 0$, let $G_i = (V_i, E_i)$ denote the subgraph of G remaining at the start of round i, and let δ_i denote an upper bound on the smallest degree of any vertex $v \in V_i$. Assume that $\delta_0 = 1$. At the start of round i, remove all vertices $v \in V_i$ such

[4]This fact holds regardless how the priority function ρ breaks ties.

that $\deg(v) \leq \max\{\delta_{i-1}, \min_{v \in V_i}\{\deg(v)\}\}$. For a vertex v removed in round i, a priority function $\rho \in \text{SL}$ is defined as $\rho(v) = \langle i, \rho_{\text{R}}(v)\rangle$ where $\rho_{\text{R}} \in \text{R}$ is a random priority function.

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

Theorem 34 *There exists a class of graphs such that for any $G = (V, E, \rho)$ in the class and for any priority function $\rho \in \text{SL}$,* JP-SL *incurs $\Omega(V)$ span and* JP-R *incurs $O(\lg V)$ span.*

Proof. Consider the algorithm to compute the priority function ρ for all vertices in a path graph G. By induction over the rounds, the graph G_i at the start of round i is a path with $|V| - 2i + 2$ vertices, and in round i the 2 vertices at the endpoints of G_i will be removed. Hence $\lceil |V|/2 \rceil$ rounds are required to assign priorities for all vertices in G. A similar argument shows that the resulting priority dag G_ρ contains a path of length $|V|/2$ along which the priorities strictly decrease. Because JP-SL trivially incurs $\Omega(1)$ span through each vertex in the longest path in G_ρ, it incurs $\Omega(V)$ span in total to color G. The span of JP-R follows from Corollary 32 and the fact that $\Delta = \Theta(1)$. $\square$

5.5 Log ordering heuristics

This section introduces the largest-log-degree-first (LLF) and smallest-log-degree-last (SLL) ordering heuristics. Given a Δ-degree graph G, we show that the expected depth of the priority dag G_ρ induced on G by a priority function $\rho \in \text{LLF}$ is $O(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta \lg V/\lg\lg V)$. The same bound applies to the depth of a priority dag G_ρ induced on a graph G by a priority function $\rho \in \text{SLL}$, though $O(\lg \Delta \lg V)$ additional span is required to calculate ρ using the method given in Figure 5-5. Combined with Theorem 28, these bounds imply that the expected span of JP-LLF is $O(\lg V + \lg \Delta(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta \lg V/\lg\lg V))$ and the expected span of JP-SLL is $O(\lg \Delta \lg V + \lg \Delta(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta \lg V/\lg\lg V))$.

The LLF ordering heuristic

The **LLF *ordering heuristic*** orders the vertices in decreasing order by the logarithm of their degree. More precisely, given a graph $G = (V, E, \rho)$ for some $\rho \in \text{LLF}$, the priority of each $v \in V$ is equal to $\rho(v) = \langle \lceil \lg(\deg(v)) \rceil, \rho_{\text{R}}(v)\rangle$, where $\rho_{\text{R}} \in \text{R}$ is a random priority function and $\lg x$ denotes $\log_2 x$.[5] For a given graph G, the following theorem bounds the depth of the priority dag G_ρ induced by $\rho \in \text{LLF}$.

Theorem 35 *Let $G = (V, E)$ be a Δ-degree graph, and let G_ρ be the priority dag induced on G by a priority function $\rho \in \text{LLF}$. The expected length of the longest directed path in G_ρ is $O(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta \lg V/\lg\lg V)$.*

Proof. Consider a k-length path $p = \langle v_1, \ldots, v_k\rangle$ in G_ρ. Let $G(\ell) \subseteq G_\rho$ be the subdag of G_ρ induced by those vertices $v \in V$ for which $\rho(v) = \lceil \lg(\deg(v)) \rceil = \ell$. Suppose that

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

$v_i \in G(\ell)$ for some $v_i \in p$. Since $\lceil \lg(\deg(v_{i-1})) \rceil \geq \lceil \lg(\deg(v_i)) \rceil$ for all $i > 1$, we have $v_{i-1} \in G(\ell')$ for some $\ell' \geq \ell$. We can therefore decompose p into a sequence of paths $p = \langle p_{\lceil \lg \Delta \rceil}, \ldots, p_0 \rangle$ such that each subpath $p_\ell \in p$ is a path through $G(\ell)$. By definition of LLF, the subdag $G(\ell)$ is a dag induced on a graph with degree 2^ℓ by a random priority function.

By Corollary 32, the expected length of p_ℓ is $O(2^\ell + \ell \lg V / \lg\lg V)$. Linearity of expectation therefore implies that

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

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

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

□

Corollary 36 *Given a graph $G = (V, E, \rho)$ for some $\rho \in$ LLF,* JP-LLF *colors all vertices in G with expected span $O(\lg V + \lg\Delta(\min\{\sqrt{E}, \Delta\} + \lg^2\Delta \lg V / \lg\lg V))$.*

Proof. The corollary follows from Theorem 28. □

The SLL ordering heuristic

To understand the **SLL *ordering heuristic***, it is convenient to consider in isolation how to compute its priority function. The pseudocode in Figure 5-5 for SLL-Assign-Priorities describes algorithmically how to perform this computation on a given graph $G = (V, E)$. As Figure 5-5 shows, a priority function $\rho \in$ SLL can be computed by iteratively removing low-degree vertices from G in rounds. The priority of a vertex $v \in V$ is the round number in which v is removed, with ties broken randomly. As with SL, SLL colors the vertices of G in the reverse order in which they are removed, but SLL-Assign-Priorities determines when to remove a vertex using a degree bound that grows exponentially. SLL-Assign-Priorities considers each degree bound for a maximum of r rounds. Effectively, a vertex is removed from G based on the logarithm of its degree in the remaining graph.

We can formalize the behavior of SLL as follows. Given a graph G, let $G_i = (V_i, E_i)$ denote the subgraph of G remaining at the start of round i. As Figure 5-5 shows, for each $d \in 0, 1, \ldots, \lg\Delta$, SLL-Assign-Priorities executes r rounds in which it removes vertices $v \in V_i$ such that $\deg(v) \leq 2^d$ in G_i.[6]

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

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

Figure 5-5: Pseudocode for SLL-ASSIGN-PRIORITIES, which computes a priority function $\rho \in \mathrm{SLL}$ for the input graph. The input parameter r denotes the maximum number of times SLL-ASSIGN-PRIORITIES is permitted to remove vertices of at most a particular degree 2^d on lines 29–36.

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

Theorem 37 *Let $G = (V, E)$ be a Δ-degree graph, and let G_ρ be the priority dag induced on G by a random priority function $\rho \in \mathrm{SLL}$. The expected length of the longest directed path in G_ρ is $O(\min\{\Delta, \sqrt{E}\} + \lg^2\Delta \lg V / \lg\lg V)$.*

PROOF. We begin with an argument similar to the proof of Theorem 35. Let $p = \langle v_1, \ldots, v_k \rangle$ be a k-length path in G_ρ, and let $G(\ell) \subseteq G_\rho$ be the subdag of G_ρ induced by those vertices $v \in V$, where $\rho(v) = \ell$. Because lines 29–36 of SLL-ASSIGN-PRIORITIES remove vertices with degree at most 2^d exactly r times for each $d \in 0, 1, \ldots, \lg \Delta$, we have that $\lfloor \rho(v)/r \rfloor = d$, and thus the degree of $G(\ell)$ is at most $2^{\lfloor \ell/r \rfloor}$. Suppose that $v_i \in G(\ell)$ for some $v_i \in p$. Because $\rho(v_{i-1}) \leq \rho(v_i)$ for all $i > 1$, we have $v_{i-1} \in G(\ell')$ for some $\ell' \geq \ell$. We can therefore decompose p into a sequence of paths $p = \langle p_{\lceil r \lg \Delta \rceil}, \ldots, p_0 \rangle$ where each $p_\ell \in p$ is a path in $G(\ell)$. By definition of SLL, the subdag $G(\ell)$ is a dag induced on a subgraph with degree at most $2^{\lfloor \ell/r \rfloor}$ by a random priority function.

By Corollary 32, the expected length of p_ℓ is $O(2^{\lfloor \ell/r \rfloor} + \lfloor \ell/r \rfloor \lg V / \lg\lg V)$. Linearity of expectation therefore implies that

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

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

CPU	Intel Xeon X5650
Clock	2.67 GHz
Hyperthreading	Disabled
Cores per processor chip	6
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	128 KiB
L3 cache/socket	12 MiB
DRAM	49 GiB
Compiler	Intel C/C++ compiler v13.1.1

Figure 5-6: Technical specification of the machine used for benchmarking.

thus conclude that

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

□

Corollary 38 *Given a graph $G = (V, E, \rho)$ for some $\rho \in$ SLL,* JP-SLL *colors all vertices in G with expected span $O(\lg \Delta \lg V + \lg \Delta(\min\{\sqrt{E}, \Delta\} + \lg^2\Delta \lg V/ \lg\lg V))$.*

Proof. The theorem follows from adding the result of Theorems 28 and 37 plus the span of SLL-Assign-Priorities. In SLL-Assign-Priorities, the span of lines 33–34 is $\lg |Q_i|$, and the expected span of lines 28–36 is therefore at most

$$\begin{aligned} \sum_{d=0}^{\lg \Delta} \left(1 + \sum_{j=1}^{r} \lg \max\{|Q_i|, 1\}\right) &= O(\lg \Delta \lg(V/(r \lg \Delta))) \\ &= O(\lg \Delta \lg V) \ . \end{aligned}$$

□

5.6 Empirical evaluation

This section evaluates the LLF and SLL ordering heuristics empirically using a suite of eight real-world and ten synthetic graphs. We describe the experimental setup used to evaluate JP-R, JP-LLF, and JP-SLL, and we compare their performance with Greedy-FF, Greedy-LF, and Greedy-SL. We compare the ordering heuristics in terms of the quality of the colorings they produce and their execution times. We conclude that LLF and SLL produce colorings with quality comparable to LF and SL, respectively, and that JP-LLF and JP-SLL scale well. We also show that the engineering quality of our implementations appears to be competitive with ColPack [154], a publicly available graph-coloring library.

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

Figure 5-7: Parameters for the generation of rMat graphs [85], where $a + b + c + d = 1$ and $b = c$, when the desired graph is undirected. An rMat graph is built by adding $|E|$ edges independently at random using the following rule: Let k be the number of 1's in a binary representation of i. As each edge is added, the probability that the ith vertex v_i is selected as an endpoint is $(a+c)^k(b+d)^{\lg n-k}$.

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

Figure 5-8: Number of edges, ratio of edges to vertices and maximum vertex degree for a collection of real-world and synthetic graphs, which lie above and below the center line, respectively.

Experimental setup

To evaluate the ordering heuristics, we implemented JP using Intel Cilk Plus [196] and engineered it to use the parallel ordering heuristics R, LLF, and SLL. To compare these parallel codes against their serial counterparts, we implemented Greedy in C to use the FF, LF, or SL ordering heuristics. In order to empirically evaluate the potential parallel performance of the serial ordering heuristics, we also engineered JP to use FF, LF, or SL. We evaluated our implementations on the machine described in Figure 5-6. Each code was compiled using `-O3` optimizations. Each measurement was taken as the median of 7 independent trials, and the averages of those measurements reported in Figures 5-9 and 5-10 were taken across 5 independent random seeds.

These implementations were run on a suite of eight real-world graphs and ten synthetic graphs. The real-world graphs came from the Large Network Dataset Collection provided by Stanford's SNAP project [250]. The synthetic graphs consist of the adversarial graphs described in Section 5.4 and a set of graphs from three classes: constant degree, 3D grid, and "recursive matrix" (rMat) [81, 85]. The adversarial graphs include cliqueChain400, a

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

Figure 5-9: Performance measurements for a set of real-world graphs taken from Stanford's SNAP project [250]. The column heading H denotes that the priority function used for the experiment in a particular row was produced by the ordering heuristic listed in the column. The average number of colors used by the corresponding ordering heuristic and graph is C_H. The time in seconds of Greedy, JP with 1 worker and with 12 workers is given by T_S, T_1 and T_{12}, respectively. Details of the experimental setup and graph suite can be found in Section 5.6.

clique-chain graph (which is illustrated in Figure 5-4) with $\Delta = 400$, and path-10M, a path graph with $|V| = 10\,\text{M}$. The constant-degree graphs — constant1M and constant500K — have 1 M and 500 k vertices and constant degrees of 100 and 200, respectively. These graphs were generated such that every pair of vertices is equally likely to be connected and every vertex has the same degree. The graph big3dgrid is a 3-dimensional grid on 10 M vertices. The rMat graphs were generated using the parameters given in Figure 5-7.

Coloring quality of R, LLF, and SLL

Figures 5-9 and 5-10 present the coloring quality of the three parallel ordering heuristics R, LLF, and SLL alongside that of their serial counterparts FF, LF, and SL.

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

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

Figure 5-10: Performance measurements for five classes of synthetically generated graphs: constant degree, rMat, 3D grid, clique chain and path. The column headings are equivalent to those in Figure 5-9.

colorings that were within 3 colors of SL for all synthetic graphs and all but 2 real-world graphs: liveJournal1 and wiki-Talk.

The liveJournal1 graph appears to benefit little from the ordering heuristics we considered. Every heuristic uses more than 300 colors, and the biggest difference between the number of colors used by any heuristic is less than 10.

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

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

Figure 5-11: Performance measurements for a set of real-world graphs taken from Stanford's SNAP project [250]. The column heading H denotes that the priority function used for the experiment in a particular row was produced by the ordering heuristic listed in the column. The average number of colors used by the corresponding ordering heuristic and graph is C_H. The time in seconds of Greedy, JP with 1 worker and with 12 workers is given by T_S, T_1 and T_{12}, respectively, where a value of ∞ indicates that the program crashed due to excessive stack usage. Details of the experimental setup and graph suite can be found in Section 5.6.

was better than LF, LLF, R, and FF by a margin of 8 to 40 colors, but not as good as SL, which used only 56 colors. These trends appear to exist, in general, for most of the graphs in the suite.

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

The parallel performance of JP was measured by computing the speedup it achieved on 12 cores and by comparing the 1-core running times of JP to an optimized serial implementation of Greedy. These results are summarized in Figures 5-9 and 5-10.

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

For comparison, Figures 5-11 and 5-12 include scalability data for JP-FF, JP-LF, and

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

Figure 5-12: Performance measurements for five classes of synthetically generated graphs: constant degree, rMat, 3D grid, clique chain and path. The column headings are equivalent to those in Figure 5-11.

JP-SL. Historically, JP-LF has been used with mixed success in practical parallel settings [12, 209, 212, 340]. Despite the fact that it offers little in terms of theoretical parallel performance guarantees, we have measured its parallel performance for our graph suite, and indeed JP-LF scales reasonably well: JP-LF$_1$/JP-LF$_{12}$ = 6.8 as compared to JP-LLF$_1$/JP-LLF$_{12}$ = 8.0 in geometric mean, not including cliqueChain400, which is omitted because JP-LF crashes on the graph due to excessive stack usage. The omission of cliqueChain400 highlights the dangers of using algorithms without good performance guarantees: it is difficult to know if the algorithm will behave badly given any particular input. In this respect, JP-FF is particularly vulnerable to adversarial inputs, as we can see by the fact that it crashes on europe-osm, which is not even intentionally adversarial. We also see this vulnerability with JP-SL, as well as generally poor scalability on the entire suite.

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

In order to validate that our implementation of GREEDY is a credible baseline, we compared it with a publicly available graph-coloring library, COLPACK [154], developed by Gebremedhin *et al.*. We found that the two implementations appeared to achieve similar performance. For example, using the SL ordering heuristic, GREEDY is 19 % faster than COLPACK in geometric-mean across the graph suite, though GREEDY is slower on 5 of the 16 graphs and as much 2.22 times slower for as-skitter.

5.7 Implementation techniques

This section describes the techniques we employed to implement JP and GREEDY for the evaluation in Section 5.6. We describe three techniques — join-trees [127], bit-vectors, and software prefetching — that improve the practical performance of JP. Where applicable, these same techniques were used to optimize the implementation of GREEDY. Overall, applying these techniques yielded 1.6 to 2.9 factor speedup for JP and a 1.2 to 1.6 factor speedup for GREEDY on the rMat-G-2M, rMat-B-2M, web-Google, and as-skitter graphs used in Section 5.6.

Join trees for reducing memory contention

Although the theoretical analysis of JP in Section 5.2 does not concern itself with contention, the implementation of JP works to mitigate overheads due to contention. The pseudocode for JP in Figure 5-2 shows that each vertex u in the graph has an associated counter $u.\mathit{counter}$. Line 17 of JP-COLOR executes a JOIN operation on $u.\mathit{counter}$. Although Section 5.2 describes how JOIN can treat $u.\mathit{counter}$ as a join counter [99] and update $u.\mathit{counter}$ using an atomic decrement-and-fetch operation, the cache-coherence protocol [309] on the machine serializes such atomic operations, giving rise to potential memory contention. In particular, memory contention may harm the practical performance of JP on graphs with large-degree vertices.

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

Bit vectors for assigning colors

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

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

Software prefetching

We used software prefetching to improve the latency of memory accesses in JP. In particular, JP uses software prefetching to mitigate the latency of the indirect memory access encountered when accessing the join trees of the successors of a vertex v on line 16 of JP-COLOR in Figure 5-2. This optimization improves T_{12} for JP by a factor of 1.2 to 1.5.

Interestingly, our implementation of GREEDY did not appear to benefit from using software prefetching in a similar context, specifically, to access the predecessors of a vertex on line 4 of GREEDY in Figure 5-1. We suspect that because GREEDY only reads the predecessors of a vertex on this line and does not write them, the processor hardware is able to generate many such reads in parallel, thereby mitigating the latency penalty introduced by cache misses.

5.8 Evaluation of serial ordering heuristics

This section summarizes our experiments with serial ordering heuristics. We compare the FF, R, LF, ID, SL, and SD heuristics in practice, both for running time and coloring quality. Our experiments indicate that the SD heuristic tends to provide colorings with higher quality than the other heuristics we have considered, confirming similar findings by Gebremedhin and Manne [153]. Although we leave the problem of devising a good parallel algorithm for SD as an open question, we devised a linear-time serial algorithm for SD, despite conjectures in the literature [94, 212] that superlinear time is required. We briefly describe our linear-time serial algorithm for SD.

Figures 5-13 and 5-14 summarizes our empirical evaluation of GREEDY run on our suite of real-world and synthetic graphs using the six ordering heuristics from Section 5.1. The measurements were taken using the same machine and methodology as was used for Figures 5-9 and 5-10. As Figures 5-13 and 5-14 show, we found that, in order, FF, R, LF, SL, and SD generally produce better colorings at the cost of greater running times. ID was outperformed in both time and quality by SL. The figure indicates that LF tends to produce better colorings than FF and R at some performance cost, and SL produces better colorings than LF at additional cost. We found that SD produces the best colorings overall, at the cost of a 4.5 geometric-mean slowdown versus SL.

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

Figure 5-13: Coloring-quality measurements for six serial ordering heuristics used by GREEDY, where measurements for real-world graphs appear above the center line and those for synthetic graphs appear below. The "*Spark*" column contains bar graphs that pictorially represent the coloring quality for each of the ordering heuristics. The height of the bar for the coloring quality C_H of ordering heuristic H is proportional to C_H. Section 5.6 details the experimental setup and graph suite used.

The SD heuristic

Figure 5-15 gives pseudocode for the GREEDY-SD algorithm, which implements the SD heuristic. Rather than trying to define a priority function for SD, the figure gives the coloring algorithm GREEDY-SD itself, since the calculation of such a priority function would color the graph as a byproduct. At any moment during the execution of the algorithm, the ***saturation degree*** of a vertex v is the number $|v.\mathit{adjColors}|$ of distinct colors of v's neighbors, and the ***effective degree*** of v is $|v.\mathit{adjUncolored}|$, its degree in the as yet uncolored graph.

The main loop of GREEDY-SD (lines 44–54) first removes a vertex v of maximum saturation degree from Q (line 45) and colors it (line 46). It then updates each uncolored neighbor $u \in v.\mathit{adjUncolored}$ of v (lines 47–52) in three steps. First, line 48 removes u from Q. Next, lines 49–50 update the set $u.\mathit{adjUncolored}$ of u's ***effective neighbors*** — u's uncolored neighbors in G — and the set $u.\mathit{adjColors}$ of colors used by u's neighbors. Finally, lines 51–52 enqueues u in Q based on u's updated information.

The crux of GREEDY-SD lies in the operation of the queue data structure Q, which is organized as an array of ***saturation tables***, each of which supports the three methods PUSHORADDKEY, POPORDELKEY, and REMOVEORDELKEY described in the caption of Figure 5-15. A saturation table can support these operations in $\Theta(1)$ time and allow its keys K to be read in $\Theta(K)$ time. At the start of each main loop iteration, entry $Q[i]$ stores the uncolored vertices in the graph with saturation degree i in a saturation table. The PUSHORADDKEY, POPORDELKEY, and REMOVEORDELKEY methods maintain the invariant that, for each table $Q[i]$, each key $j \in \text{KEYS}(Q[i])$ is associated with a nonempty set of vertices, such that each vertex $v \in Q[i][j]$ has saturation degree i and effective degree j.

The following theorem shows that GREEDY-SD runs in linear time.

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

Figure 5-14: Running-time measurements for six serial ordering heuristics used by GREEDY, where measurements for real-world graphs appear above the center line and those for synthetic graphs appear below. The "*Spark*" column contains bar graphs that pictorially represent the serial running time for each of the ordering heuristics. The height of the bar for the serial running time T_S of ordering heuristic H is proportional to $\log T_S$. Section 5.6 details the experimental setup and graph suite used.

Theorem 39 GREEDY-SD *colors a graph* $G = (V, E)$ *according to the* SD *ordering heuristic in* $\Theta(V + E)$ *time.*

PROOF. The operations PUSHORADDKEY, POPORDELKEY, and REMOVEORDELKEY each take $\Theta(1)$ time, and a given saturation table's key set K can be read in $\Theta(K)$ time. Line 45 can thus find a vertex v with maximum saturation degree s in $\Theta(|\text{KEYS}(Q[s])|)$ time. Line 46 can color v in $\Theta(\deg(v))$ time, and lines 52–54 maintain s in $\Theta(s)$ time. Because $s + |\text{KEYS}(Q[s])| \leq \deg(v)$, lines 44–54 evaluate v in $\Theta(\deg(v))$ time. The handshaking lemma [100, p. 1172–3] implies the theorem, because each vertex in V is evaluated once. □

5.9 Related work

Parallel coloring algorithms have been explored extensively in the distributed computing domain [13, 32, 162, 163, 209, 226, 227, 254]. These algorithms are evaluated in the message-passing model, where nodes are allowed unlimited local computation and exchange messages through a sequence of synchronized rounds. Kuhn [226] and Barenboim and Elkin [32] independently developed $O(\Delta + \lg^* n)$-round message passing algorithms to compute a deterministic greedy coloring.

Several greedy coloring algorithms have been described in synchronous PRAM models. Goldberg *et al.* [162] describe an algorithm for finding a greedy coloring of $O(1)$-degree graphs in $O(\lg n)$ time in the EREW PRAM model using a linear number of processors. They observe that their technique can be applied recursively to color Δ-degree graphs in $O(\Delta \lg \Delta \lg n)$ time. Their strategy incurs $\Omega(\lg \Delta(V + E))$ (superlinear) work, however.

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

Figure 5-15: The GREEDY-SD algorithm computes a coloring for the input graph $G = (V, E)$ using the SD heuristic. Each uncolored vertex $v \in V$ maintains a set $v.\mathit{adjColors}$ of colors used by its neighbors and a set $v.\mathit{adjUncolored}$ of uncolored neighbors of v. The PUSHORADDKEY method adds a specified key, if necessary, and then adds an element to that key's associated set. The POPORDELKEY and REMOVEORDELKEY methods remove an element from a specified key's associated set, deleting that key if the set becomes empty. The variable s maintains the maximum saturation degree of G.

Catalyurek *et al.* [81] present the algorithm ITERATIVE, which first speculatively colors a graph G and then fixes coloring conflicts, that is, corrects the coloring where two adjacent vertices are assigned the same color. The process of fixing conflicting colors can introduce new conflicts, though the authors observe empirically that comparatively few iterations suffice to find a valid coloring. We ran ITERATIVE on our test system and found that JP-LLF uses 13% fewer colors and takes 19% less time in geometric mean of number of colors and relative time, respectively, over all graphs in our test suite. Furthermore, we found that JP-SLL uses 17% fewer colors, but executes in twice the time of ITERATIVE. We do not know the extent to which the optimizations enjoyed by our algorithms could be adopted by speculative-coloring algorithms, however, and so it is likely too soon to draw conclusions about comparisons between the strategies.

5.10 Conclusion

Because of the importance of graph coloring, considerable effort has been invested over the years to develop ordering heuristics for serial graph-coloring algorithms. For the traditional "serial" LF and SL ordering heuristics, we have developed "parallel" analogs — the LLF and SLL heuristics, respectively — which approximate the traditional orderings, generating colorings of comparable quality while offering provable guarantees on parallel scalability. The correspondence between serial ordering heuristics and their parallel analogs is fairly direct for LF and LLF. LLF colors any two vertices whose degrees differ by more than a factor of 2 in the same order as LF. In this sense, LLF can be viewed as a simple coarsening of the vertex ordering used by LF. Although SLL is inspired by SL, and both heuristics tend to color vertices of smaller degree later, the correspondence between SL and SLL is not as straightforward. We relied on empirical results to determine the degree to which SLL captures the salient properties of SL. The LLF and SLL heuristics thus allow programmers to improve the colorings produced by parallel greedy graph-coloring programs, as they might for serial graph-coloring programs, without sacrificing parallel scalability. These heuristics thus support a principled approach to improving the coloring quality of parallel coloring algorithms.

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

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

5.11 Recent developments

Hasenplaugh improved the bounds on the span of JP-R, JP-LLF, and JP-SLL, reducing the $O(\lg^3\Delta \lg V / \lg\lg V)$ term in each bound to $O(\lg^2\Delta \lg V / \lg(e \lg V/D))$ [176].

Chapter 6

Deterministic Parallel Random-Number Generation

This chapter presents the pedigree mechanism and DotMix deterministic parallel random-number generator [249]. This work was conducted in collaboration with Charles E. Leiserson and Jim Sukha. This chapter presents an improved analysis of DotMix compared to the analysis that originally appeared in [249].

6.1 Introduction

Dynamic multithreading concurrency platforms fail to encapsulate an important source of nondeterminism for applications that employ (pseudo)random number generators (RNG's). RNG's are useful for randomized algorithms [289], which provide efficient solutions to a host of combinatorial problems, and are essential for Monte Carlo simulations, which consume a large fraction of computing cycles [273] for applications such as option pricing, molecular modeling, quantitative risk analysis, and computer games. Unfortunately, typical implementations of RNG's are either nondeterministic or exhibit high overheads when used in dynamic multithreaded code. As a result, performance engineers are left to contend either with slow code or with nondeterministic program behaviors that can be difficult to elicit repeatably and reliably. Nondeterminism undermines a programmer's ability to understand what the program does and to find and eradicate incorrect behaviors.

To understand the problem that RNG's introduce in dynamic multithreaded programs, we first review conventional serial RNG's and consider how they are traditionally adapted for use in parallel programs. Then we examine the ramifications of this adaptation on dynamic multithreaded programs.

A serial RNG operates as a stream. The RNG begins in some initial state S_0. The ith request for a random number updates the state S_{i-1} to a new state S_i, and then it returns some function of S_i as the ith random number. One can construct a parallel RNG using a serial RNG, but at the cost of introducing nondeterminism. One way that a serial RNG can be used directly in a dynamic multithreaded application is as a global RNG where the stream's update function is protected by a lock. This strategy introduces nondeterminism in the order of lock acquisition, however, as well as contention on the lock that can adversely affect performance.

A more practical alternative that avoids lock contention is to use ***worker-local RNG's***, that is, to construct a parallel RNG by having each worker thread maintain its own serial

RNG for generating random numbers. Unfortunately, this solution fails to eliminate nondeterminism when applied to dynamic multithreaded programs. In particular, the underlying nondeterministic scheduler might execute a given call to the RNG on different workers during different runs of the program, even if the sequence of random numbers produced by each worker is deterministic.

Although ***deterministic parallel random-number generators (DPRNG's)*** exist for Pthreading platforms, they are ineffective for dynamic multithreading platforms. For example, SPRNG [273] is an excellent DPRNG which creates independent RNG's via a parameterization process.[1] For a few Pthreads that are spawned at the start of a computation and which operate independently, SPRNG can produce the needed RNG for each Pthread. For a dynamic multithreaded program, however, which may contain millions of ***strands*** — serial sequences of executed instructions containing no parallel control — each strand might need its own RNG, and SPRNG cannot cope.

Consider, for example, a program that uses SPRNG to generate a random number at each leaf of the computation of a parallel, exponential-time, recursive Fibonacci calculation, `fib`. Every time `fib` spawns a recursive subcomputation, a new strand is created, and the program calls SPRNG to produce a new serial RNG stream from the existing serial RNG. The `fib` program is deterministic, since each strand receives the same sequence of random numbers in every execution. In an implementation of this program, however, we observed two significant problems:

- When computing `fib(21)`, the program using SPRNG was almost 50,000 times slower than a nondeterministic version that maintains worker-local Mersenne twister [274] RNG's from the GNU Scientific Library [147].
- SPRNG's default RNG only guarantees the independence of 2^{19} streams, and computing `fib(n)` for `n` > 21 forfeits this guarantee.

Of course, SPRNG was never intended for this kind of use case where many streams are created with only a few random numbers generated from each stream. This example does show, however, the inadequacy of a naive solution to the problem of deterministic parallel random-number generation for dynamic multithreading platforms.

Contributions

In this chapter, we investigate the problem of deterministic parallel random-number generation for dynamic multithreading platforms. In particular, this chapter makes the following contributions:

- A compiler and runtime-system mechanism, called "pedigrees," for tracking the "lineage" of each strand in a dynamic multithreaded program. Pedigrees introduce negligible overhead across a suite of 10 MIT Cilk [146] benchmark applications.
- A general strategy for efficiently generating quality deterministic parallel random numbers based on compressing the strand's pedigree and "mixing" the result.
- A high-quality DPRNG library for Cilk Plus, called DotMix, which is based on compressing the pedigree via a dot-product [116] and "RC6-mixing" [98,332] the result. The statistical quality of the numbers generated by DotMix appears to rival that of the popular Mersenne twister RNG [274]. The cost of calling DotMix depends on the depth of nested spawns in a Cilk program, which tends to be small for programs

[1]In addition to parametrized streams, other approaches to parallelizing RNG's exist, such as leapfrogging and splitting (for a survey, see [92]), and many of these parallel RNG schemes can also be adapted to create similar DPRNG's for Pthreaded programs.

Cilk programs with good scalability. For realistic benchmark applications that use pseudorandom numbers, the "price of determinism" paid for using DotMix instead of worker-local instances of Mersenne twister is at most 21 %, and sometimes much less.

Pedigrees and DotMix enable programmers to write randomized dynamic multithreaded programs whose behavior is deterministic for any fixed seed of the RNG. As a consequence, the semantics of such a program are serial, meaning that programmers can understand what the program does from its serial execution. Furthermore, as Section 6.5 will show, the interface to DotMix mirrors that of an ordinary serial RNG, meaning that programmers need only minimal changes to make a program use DotMix instead of an ordinary serial RNG. These properties can simplify the task of understanding what randomized dynamic-multithreaded programs do, which helps programmers engineer these codes in a principled fashion. Finally, the performance of DotMix ensures that programmers need not pay undue performance to enjoy this deterministic behavior. In particular, the "price of determinism" for DotMix seems amply fast for debugging purposes, which is a major reason for desiring repeatability.

Outline

The remainder of this chapter is organized as follows. Section 6.2 defines pedigrees and describes how they can be incorporated into a dynamic multithreading platform. Section 6.3 presents the DotMix DPRNG, showing how pedigrees can be leveraged to implement DPRNG's. Section 6.4 describes other pedigree-based DPRNG schemes, focusing on one based on linear congruential generators [222]. Section 6.5 presents a programming interface for a DPRNG library. Section 6.6 presents performance results measuring the overhead of runtime support for pedigrees in MIT Cilk, as well as the overheads of DotMix in Cilk Plus on synthetic and realistic applications. Section 6.7 describes related work, and Section 6.8 offers some concluding remarks.

6.2 Pedigrees

A pedigree scheme uniquely identifies each strand of a dynamic multithreaded program in a scheduler-independent manner. This section introduces "spawn pedigrees," a simple pedigree scheme that can be easily maintained by a dynamic multithreading runtime system. We describe the changes that Intel implemented in their Cilk Plus concurrency platform to implement spawn pedigrees. Their runtime support provides an application programming interface (API) that allows user programmers to access the spawn pedigree of a strand, which can be used to implement a pedigree-based DPRNG scheme. We finish by describing an important optimization for parallel loops, called "flattening."

We shall focus on dialects of Cilk to contextualize our discussion, because we used Cilk platforms to implement the spawn-pedigree scheme and study its empirical behavior. The runtime support for pedigrees that we describe can be adapted to other dynamic multithreading platforms, however, as Section 6.8 will discuss. To simplify the "spawn pedigree" scheme, in this section, we shall not assume that strands respect function boundaries, meaning that a single strand might contain executed instructions from multiple function frames.

Pedigree schemes

Pedigrees are deterministic labels for the executed instructions in a dynamic multithreaded

```
01 int main(void) {
02   int x = fib(4);
03   printf("x = %d\n", x);
04   return x;
05 }
06
07 int fib(int n) {
08   if (n < 2) return n;
09   else {
10     int x, y;
11     x = cilk_spawn fib(n-1);
12     y = fib(n-2);
13     cilk_sync;
14     return x+y;
15   }
16 }
```

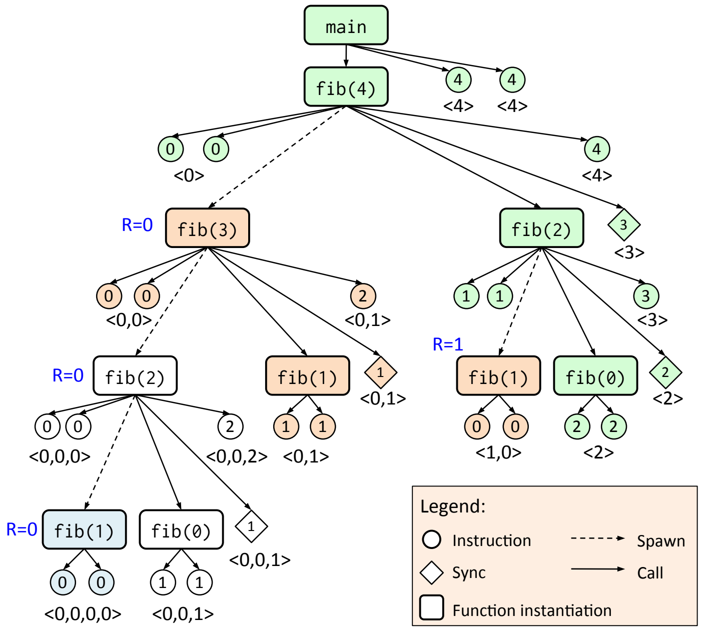


Figure 6-1: Cilk code for a recursive Fibonacci calculation and the invocation tree for an execution of `fib(4)`. Pedigrees are labeled for each instruction. For example, `cilk_sync` instruction in `fib(4)` has pedigree $\langle 3 \rangle$. A left-to-right preorder traversal of the tree represents the serial execution order. For example, for the children of the node for `fib(4)`, the first two instructions with rank 0 correspond to lines 8 and 10 from Figure 6-1, the subtree rooted at node `fib(4)` between `fib(3)` and the sync node corresponds to the execution of the sync block (lines 11–13), and the last instruction with rank 4 corresponds to the return in line 14. Instructions and functions are labeled with their ranks. For example, `fib(3)` has a rank of 0.

program execution that partition the instructions into valid strands. For the remainder of this section, assume that the dynamic multithreaded program in question would be deterministic if each RNG call in the program always returned the same random number on every execution. For such computations, a pedigree scheme maintains two useful properties:

1. ***Schedule independence***: For any instruction x, the value of the pedigree for x, denoted $J(x)$, does not depend on how the program is scheduled on multiple processors.
2. ***Strand uniqueness***: All instructions with the same pedigree form a strand.

Together, Properties 1 and 2 guarantee that pedigrees identify strands of a dynamic multithreaded program in a deterministic fashion, regardless of scheduling. Therefore, one can generate a random number for each strand by simply hashing its pedigree.

The basic idea of a pedigree scheme is to name a given strand by the path from the root of the ***invocation tree*** — the tree of function (instances) where F is a ***parent*** of G, denoted $F = \texttt{parent}(G)$, if F spawns or calls G. Imagine labeling each instruction of a function with a ***rank***, which is the number of calls, spawns, or syncs that precede it in the function. Then the pedigree of an instruction x can be encoded by giving its rank and a list of ancestor ranks. For example, the instruction x might have rank 3 and be the 5th child of the 1st child of the 3rd child of the 2nd child of the root, and thus its pedigree would be $J(x) = \langle 2, 3, 1, 5, 3 \rangle$. Such a scheme satisfies Property 1, because the invocation tree is the same no matter how the computation is scheduled. It also satisfies Property 2, because two instructions with the same pedigree cannot have a spawn or sync between them.

Spawn pedigrees improve on this simple scheme by defining ranks using only spawns and syncs, omitting calls and treating called functions as being "inlined" in their parents.

We can define spawn pedigrees operationally in terms of a serial execution of a dynamic multithreaded program. The runtime system conceptually maintains a stack of ***rank counters***, where each rank counter corresponds to an instance of a spawned function. Program execution begins with a single rank counter with value 0 on the stack for the root (`main`) function F_0. Three events cause the rank-counter stack to change:

1. On a **spawn** of a function G, push a new rank counter with value 0 for G onto the bottom of the stack.
2. On a return from the **spawn** of G, pop the rank counter (for G) from the bottom of the stack, and then increment the rank counter at the bottom of the stack.
3. On a **sync** statement inside a function F, increment the rank counter at the bottom of the stack.

For any instruction x, the pedigree $J(x)$ is simply the sequence of ranks on the stack when x executes. Figure 6-1 shows the Cilk code for a recursive Fibonacci calculation and the corresponding invocation tree for an execution of `fib(4)` with spawn pedigrees labeled on instructions. Intuitively, the counter at the bottom of the rank-counter stack tracks the rank of the currently executing instruction x with respect to the spawned ancestor function closest to x. Thus, the increment at the bottom of the stack occurs whenever resuming the continuation of a **spawn** or a **sync** statement. This operational definition of spawn pedigrees satisfies Property 2, because an increment occurs whenever any parallel control is reached and the values of the pedigrees are strictly increasing according to a lexicographic order. Because a spawn pedigree is dependent only on the invocation tree, spawn pedigrees also satisfy Property 1.

Runtime support for spawn pedigrees

Supporting spawn pedigrees in parallel in a dynamic multithreaded program is simple but subtle. Let us first acquire some terminology. We extend the definition of "parent" to instructions, where for any instruction x, the ***parent*** of x, denoted parent(x), is the function that executes x. For any nonroot function F, define the ***spawn parent*** of F, denoted spParent(F), as parent(F) if F was spawned, or spParent(parent(F)) if F was called. Intuitively, spParent(F) is the closest proper ancestor of F that is a spawned function. Define the ***spawn parent*** of an instruction x similarly: spParent(x) = spParent(parent(x)). The ***rank*** of an instruction x, denoted $R(x)$, corresponds to the value in the bottom-most rank counter at the time x is executed in a serial execution, and each more-distant spawn parent in the ancestry of x directly maps to a rank counter higher in the stack.

The primary complication for maintaining spawn pedigrees during a parallel execution is that, while one worker p is executing an instruction x in F = spParent(x), another worker p' might steal a continuation in F and continue executing, conceptually modifying the rank counter for F. To eliminate this complication, when p spawns a function G from F, it saves $R(G)$ — the rank-counter value of F when G was spawned — into the frame of G, thereby guaranteeing that any query of the pedigree for x has access to the correct rank, even if p' has resumed execution of F and incremented its rank counter.

Figure 6-2 presents an API that allows a currently executing strand s to query its spawn pedigree. For any instruction x belonging to a strand s, this API allows s to walk up the chain of spawned functions along the x-to-root path in the invocation tree and access the appropriate rank value for x and each ancestor spawned function. The sequence of ranks discovered along this walk is precisely the reverse of the pedigree $J(x)$.

We persuaded Intel to modify its Cilk Plus concurrency platform [196] to include pedi-

Function	*Description*	*Implementation*
RANK()	Returns $R(x)$.	Returns $p \rightarrow rank$
SPPARENT()	Returns spParent(x).	Returns $p \rightarrow current\_frame$.
RANK$(\widehat{F})$	Returns $R(\widehat{F})$.	Returns $\widehat{F} \rightarrow brank$
SPPARENT$(\widehat{F})$	Returns spParent$(\widehat{F})$.	Returns $\widehat{F} \rightarrow parent$
STRANDBREAK()	Ends the currently-executing strand.	$p \rightarrow rank{+}{+}$

Figure 6-2: An API for spawn pedigrees in Cilk Plus. In these operations, x is the currently executing instruction for a worker p, and $\widehat{F}$ is a spawn-wrapper function which is an ancestor of x in the computation tree. These operations allow a worker to walk up the computation tree to compute $J(x)$. A worker can also call STRANDBREAK() to end its currently executing strand.

grees. The Intel C/C++ compiler with Cilk Plus compiles the spawning of a function G as a call to a ***spawn-wrapper*** function $\widehat{G}$, which performs the necessary runtime manipulations to effect the spawn, one step of which is calling the function G. Thus, for any function G, we have spParent$(G) = \widehat{G}$, and for any instruction x, the pedigree $J(x)$ has a rank counter for each spawn-wrapper ancestor of x.

Implementing this API in Cilk Plus requires additional storage in spawn-wrapper frames and in the state of each worker thread. For every spawned function F, the spawn wrapper $\widehat{F}$ stores the following rank information in F's frame:

- $\widehat{F} \rightarrow brank$: A 64-bit[2] value that stores $R(F)$.
- $\widehat{F} \rightarrow parent$: The pointer to spParent$(\widehat{F})$.

In addition, every worker p maintains two values in worker-local storage for its currently executing instruction x:

- $p \rightarrow current\_frame$: the pointer to spParent(x).
- $p \rightarrow rank$: a 64-bit value storing $R(x)$.

As Figure 6-2 shows, to implement the API, the runtime system reads these fields to report a spawn pedigree. In terms of the operational definition of spawn pedigrees, the *rank* field in p holds the bottom-most rank counter on the stack for the instruction x that p is currently executing.

To maintain these fields, the runtime system requires additional storage to save and restore the current spawn-parent pointer and rank counter for each worker whenever it enters or leaves a nested spawned function. In particular, we allocate space for a rank and parent pointer in the stack frame of every ***Cilk function*** — function that can contain **spawn** and **sync** statements:

- $G \rightarrow rank$: a 64-bit value that stores $R(x)$ for some instruction x with spParent$(x) = G$.
- $G \rightarrow sp\_rep$: the pointer to spParent(G).

These fields are only used to save and restore the corresponding fields for a worker p. Whenever p is executing a Cilk function G which spawns a function F, it saves its fields into G before beginning execution of F. When a worker p' (which might or might not be p) resumes the continuation after the **spawn** statement, p' restores its values from G. Similarly, saving and restoring also occurs when a worker stalls at a **sync** statement. Figure 6-3 summarizes the runtime operations needed to maintain spawn pedigrees.

Although the implementation of spawn pedigrees in Intel Cilk Plus required changes to the Intel compiler, ordinary C/C++ functions need not be recompiled for the pedigree scheme to work. The reason is that the code in Figure 6-3 does not perform any operations

[2]A 64-bit counter never overflows in practice, since 2^{64} is a *big* number.

```
On a spawn of $F$ from $G$:
1  $G \rightarrow rank = p \rightarrow rank$
2  $G \rightarrow sp\_rep = p \rightarrow current\_frame$
3  $\widehat{F} \rightarrow brank = G \rightarrow rank$
4  $\widehat{F} \rightarrow parent = G \rightarrow sp\_rep$
5  $p \rightarrow rank = 0$
6  $p \rightarrow current\_frame = \widehat{F}$
```

```
On stalling at a sync in $G$:
7  $G \rightarrow rank = p \rightarrow rank$
```

```
On resuming the continuation of a spawn or sync in $G$:
8  $p \rightarrow rank = G \rightarrow rank$++
9  $p \rightarrow current\_frame = G \rightarrow sp\_rep$
```

Figure 6-3: How a worker p maintains spawn pedigrees. In this pseudocode, G denotes a Cilk function instantiation, F denotes the instantiation of a function that G spawns, and $\widehat{F}$ denotes the instantiation of the spawn wrapper of F. The value $p \rightarrow current\_frame$ need not be saved into $G \rightarrow sp\_rep$, because the first **spawn** in G will have saved this value already, and this value is fixed for G.

on entry or exit to called functions. Consequently, the scheme works even for programs that incorporate legacy and third-party C/C++ binaries.

To implement DPRNG's, it is useful to extend the API in Figure 6-2 to include a StrandBreak function that allows the DPRNG to end a currently executing strand explicitly. In particular, if a user requests multiple random numbers from a DPRNG in a serial sequence of instructions, then the DPRNG can let each call to get a random number terminate a strand in that sequence using this function, meaning that the DPRNG produces at most one random number per strand. Like a **spawn** or **sync**, when a worker p encounters a StrandBreak call, the next instruction after the StrandBreak that p executes is guaranteed to be part of a different strand, and thus have a different pedigree. The StrandBreak function is implemented by incrementing $p \rightarrow rank$.

Pedigree flattening for parallel loops

As an optimization, we can simplify spawn pedigrees for parallel loops. The Cilk Plus runtime system implements its parallel loop construct, `cilk_for`, using a balanced binary recursion tree implemented with `cilk_spawn`'s and `cilk_sync`'s, where each leaf performs a chunk of iterations serially. Rather than track ranks at every level of this recursion tree, the Cilk Plus pedigree scheme conceptually "cuts out the middle man" and ***flattens*** all the iterations of the `cilk_for` loop so that they share a single level of pedigree. The basic idea is simply to let the rank of an iteration be the loop index. Consequently, iterations in a `cilk_for` can be referenced within the `cilk_for` by a single value, rather than a path through the binary recursion tree. To ensure that `cilk_spawn` and `cilk_sync` statements within a loop iteration do not affect the pedigrees of other loop iterations, the body of each loop iteration is treated as a spawned function with respect to its pedigree. This change simplifies the pedigrees generated for `cilk_for` loops by reducing the effective spawn depth of strands within the `cilk_for` and, as Section 6.6 shows, the cost of reading the pedigree as well.

6.3 DotMix: A pedigree-based DPRNG

This section presents DotMix, a high-quality statistically random pedigree-based DPRNG. DotMix operates by hashing the pedigree and then "mixing" the result. We investigate theoretical principles behind the design of DotMix, and in particular, we show that DotMix is 2-universal [80]. These properties offer evidence that pedigree-based DPRNG's can generate pseudorandom numbers of high quality for real applications. We also examine empirical test results using Dieharder [71], which suggest that DotMix generates high-quality random numbers in practice.

The DotMix *DPRNG*

At a high level, DotMix generates random numbers in two stages. First, DotMix compresses the pedigree into a single machine word while attempting to maintain uniqueness of compressed pedigrees. Second, DotMix "mixes" the bits in the compressed pedigree to produce a pseudorandom value.

To describe DotMix formally, let us first establish some notation. We assume that our computer has a word width of w bits. We choose a prime $p < m = 2^w$ and assume that each rank j_i in the pedigree falls in the range $1 \leq j_i < p$. Our library implementation of DotMix simply increments each rank in the spawn-pedigree scheme from Section 6.2 to ensure that ranks are nonzero. Let $\mathbb{Z}_m$ denote the universe of (unsigned) w-bit integers over which calculations are performed, and let $\mathbb{Z}_p$ denote the finite field of integers modulo p. Consequently, we have $\mathbb{Z}_p \subseteq \mathbb{Z}_m$. We assume that the spawn depth $d(x)$ for any instruction x in a dynamic multithreaded program is bounded by $d(x) \leq D$.[3] A pedigree $J(x)$ for an instruction x at spawn depth $d(x)$ can then be represented by a D-length vector $J(x) = \langle j_1, j_2, \ldots, j_D \rangle \in \mathbb{Z}_p^D$, where $j_i = 0$ for $D - d(x)$ entries. Which entries are padded out with 0 does not matter for the theorems that follow, as long as distinct pedigrees remain distinct. In our implementation, we actually pad out the first entries.

For a given pedigree $J \in \mathbb{Z}_p^D$, the random number produced by DotMix is the hash $h(J)$, where $h(J)$ is the composition of

1. a ***compression function*** $c : \mathbb{Z}_p^D \to \mathbb{Z}_p$ that hashes each pedigree J into a single integer $c(J)$ less than p, and
2. a ***mixing function*** $\mu : \mathbb{Z}_m \to \mathbb{Z}_m$ that "mixes" the compressed pedigree value $c(J)$.

Let us consider each of these functions individually.

The goal of a compression function c is to hash each pedigree J into an integer in $\mathbb{Z}_p$ such that the probability of a ***collision*** — two distinct pedigrees hashing to the same integer — is small. To compress pedigrees, DotMix computes a dot product of the pedigree with a vector of random values [116]. More formally, DotMix uses a compression function c chosen uniformly at random from the following hash family.[4]

Definition 1 *Let $\Gamma = \langle \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_D \rangle$ be a vector of integers chosen uniformly at random from $\mathbb{Z}_p^{D+1}$. Define the compression function $c_\Gamma : \mathbb{Z}_p^{D+1} \to \mathbb{Z}_p$ by*

$$c_\Gamma(J) = \gamma_0 + \left(\sum_{k=1}^{D} \gamma_k j_k \right) \bmod p \ ,$$

where $J = \langle j_1, j_2, \ldots, j_D \rangle \in \mathbb{Z}_p^D$. The **DotMix** ***compression-function family*** *is the set*

$$C_{\text{DotMix}} = \left\{ c_\Gamma : \Gamma \in \mathbb{Z}_p^{D+1} \right\} \ .$$

[3]Although a dynamic multithreaded program might have arbitrarily deeply nested spawns in theory, this depth imposes a lower bound on the span of a computation, and scalable programs generally have low depth. A reasonable upper bound in practice is $D = 100$.

[4]The definition of C_{DotMix} presented here differs from that in the original paper on this work [249], but represents the implementation of DotMix more faithfully.

The next theorem justifies that the probability is small that a randomly chosen compression function $c_\Gamma \in C_{\text{DotMix}}$ causes two distinct pedigrees to collide. In fact, we show a stronger result, namely, that the DotMix compression-function family is 2-independent [402]. [5]

Theorem 40 *Let $c_\Gamma \in C_{\text{DotMix}}$ be a randomly chosen compression function, and let h and h' be arbitrary integers in $\mathbb{Z}_p$. Then, for any two distinct pedigrees $J, J' \in \mathbb{Z}_p^D$, we have* $\Pr\{c_\Gamma(J) = h \cap c_\Gamma(J') = h'\} = 1/p^2$.

Proof. We first show that $\Pr\{c_\Gamma(J) = h\} = 1/p$. Let $J = \langle j_1, j_2, \ldots, j_D\rangle$. Then we have (modulo p) that

$$h = \gamma_0 + \sum_{k=1}^{D} \gamma_i j_i \ ,$$

which implies that

$$h - \sum_{k=1}^{D} \gamma_i j_i = \gamma_0 \ .$$

Because γ_0 is chosen uniformly at random from $\mathbb{Z}_p$, the probability that it equals any particular value of $h - \sum_{k=1}^{D} \gamma_i j_i$ is $1/p$.

We now show that the conditional probability $\Pr\{c_\Gamma(J') = h' \mid c_\Gamma(J) = h\}$ is $1/p$. Let $J = \langle j_1, j_2, \ldots, j_D\rangle$, and let $J' = \langle j'_1, j'_2, \ldots, j'_D\rangle$. Because $J \neq J'$, there must exist some index k in the range $1 \leq k \leq D$ such that $j_k \neq j'_k$. Without loss of generality, assume that $k = 1$. Then the difference between h and h' equals the following sum (modulo p):

$$\begin{aligned} h - h' &= c_\Gamma(J) - c_\Gamma(J') \\ &= \gamma_1 j_1 - \gamma_1 j'_1 + \sum_{k=2}^{D} \gamma_k j_k - \sum_{k=2}^{D} \gamma_k j'_k \ . \end{aligned}$$

Rearranging terms gives us

$$(j_1 - j'_1)\gamma_1 = h - h' + \sum_{k=2}^{D} \gamma_k (j'_k - j_k) \ .$$

Consider fixed values for J, J', and $\gamma_2, \ldots, \gamma_D$. Let $a = j_1 - j'_1 \neq 0$, let $x = \gamma_1$, and let $y = h - h' + \sum_{k=2}^{D} \gamma_k (j'_k - j_k)$. The equation above can thus be rewritten simply as $y = ax$.

We argue that for any fixed choice of $y \in \mathbb{Z}_p$ and nonzero $a \in \mathbb{Z}_p$, there is exactly one choice of $x \in \mathbb{Z}_p$ such that $y = ax$, namely, $x = a^{-1}y$. For the sake of contradiction, suppose that there are two distinct values x_1 and x_2 such that $y = ax_1 = ax_2$. This supposition implies that $0 = ax_1 - ax_2 = a(x_1 - x_2)$ modulo p, which is satisfied if and only if either $a = 0$ or $x_1 - x_2 = 0$, because p is prime. Because $a \neq 0$, we must have $x_1 - x_2 = 0$, contradicting the supposition that x_1 and x_2 are distinct. Therefore, there is one value of x satisfying $y = ax$. Because $x = \gamma_1$ is a randomly chosen value from $\mathbb{Z}_p$, the probability that x satisfies $y = ax$ is $1/p$.

The theorem follows from multiplying $\Pr\{c_\Gamma(J) = h\}$ and $\Pr\{c_\Gamma(J') = h' \mid c_\Gamma(J) = h\}$.
□

A corollary of Theorem 40 is that the probability that any two distinct pedigrees collide

[5] The original paper on this work [249] only shows that this compression-function family is 2-universal.

is $1/p$. This low probability of collision allows DotMix to generate many random numbers with a low probability that any pair collide. By Boole's Inequality [100, p. 1195], given n distinct pedigrees and using a random function from C_{DotMix}, the probability of a collision among any of their compressed pedigrees is at most $\binom{n}{2}(1/p) = n(n-1)/2p$. For example, our DotMix implementation uses the prime $p = 2^{64} - 59$, for which the probability that hashing 5 million pedigrees results in a collision in their compressed values is less than 1 in a million.

Although the compression function effectively hashes a pedigree into an integer less than p with a small probability of collision, two similar pedigrees may yet have "similar" hash values, whereas we would like them to be statistically "dissimilar." In particular, for a given compression function c_Γ, two pedigrees that differ only in their kth coordinate differ in their compressions by a predictable multiple of γ_k. To reduce the statistical correlation in generated random values, DotMix "mixes" the bits of a compressed pedigree using a mixing function based on the RC6 block cipher [98, 332]. For any w-bit input z, where we assume that w is even, let $\phi(z)$ denote the function that swaps the high- and low-order $w/2$ bits of z, that is,

$$\phi(z) = \left\lfloor \frac{z}{\sqrt{m}} \right\rfloor + \sqrt{m}\left(z \bmod \sqrt{m}\right) ,$$

and let

$$f(z) = \phi(2z^2 + z) \bmod m .$$

DotMix uses the mixing function $\mu(z) = f^{(r)}(z)$, which applies r rounds of $f(z)$ to the compressed pedigree value z. Contini *et al.* [98] prove that $f(z)$ is a one-to-one function, and hence, for two distinct pedigrees J and J', the probability that $\mu(c(J)) = \mu(c(J'))$ is $1/p$, unchanged from Theorem 40.

DotMix allows a seed to be incorporated into the hash of a pedigree. The random number generated for a pedigree J is actually the value of a hash function $h(J, \sigma)$, where σ is a seed. Such a seed may be incorporated into the computation of $\mu(c_\Gamma(J))$ in several ways. For instance, we might XOR or otherwise combine the seed with the result of $\mu(c_\Gamma(J))$, computing, for example, $h(J, \sigma) = \sigma \oplus \mu(c_\Gamma(J))$. This scheme does not appear to be particularly good, because it lacks much statistical variation between the numbers generated by one seed versus another. A better scheme, which DotMix adopts, is to combine the seed with the compressed pedigree *before* mixing. In particular, DotMix incorporates the seed in selecting a hash function from the DotMix compression-function family C_{DotMix}.

The statistical quality of **DotMix**

Although DotMix is not a cryptographically secure RNG, it appears to generate high-quality random numbers as evinced by Dieharder [71], a collection of statistical tests designed to empirically test the quality of serial RNG's. Figure 6-4 summarizes the Dieharder test results for DotMix and compares them to those of the Mersenne twister [274], whose implementation is provided in the GNU Scientific Library [147]. As Figure 6-4 shows, with 2 or more iterations of the mixing function, DotMix generates random numbers of comparable quality to Mersenne twister. In particular, Mersenne twister and DotMix with 2 or more mixing iterations generally fail the same set of Dieharder tests. Because the Dieharder tests are based on P-values [184], it is not surprising to see statistical variation in the number of "Weak" and "Poor" results even from high-quality RNG's. We report the median of 5 runs using 5 different seeds to reduce this variation.

Test	*r*	*Passed*	*Weak*	*Poor*	*Failed*
Mersenne twister	—	79	7	7	14
DotMix (tree)	16	83	6	4	14
	8	84	6	4	13
	4	81	5	7	14
	2	81	5	5	16
	1	3	2	3	99
	0	0	0	0	107
DotMix (loop)	16	82	2	8	15
	8	79	6	8	14
	4	79	5	8	15
	2	79	4	8	16
	1	55	2	8	42
	0	2	0	1	104
LCGMix (tree)	4	84	4	6	13
	0	24	6	21	56

Figure 6-4: A summary of the quality of DotMix on the Dieharder tests compared to the Mersenne twister. For the entries labeled "tree," DotMix generates 3^{20} random numbers in a parallel divide-and-conquer ternary tree fashion using spawns. For the entries labeled "loop," a `cilk_for` loop generates 3^{20} random numbers. The column labeled r indicates the number of mixing iterations. Each successive column counts the number of tests that produced the given status, where the status of each test was computed from the median of 5 runs of the generator using 5 different seeds. The table also summarizes the Dieharder test results for LCGMix.

When using Dieharder to measure the quality of a parallel RNG, we confronted the issue that Dieharder is really designed to measure the quality of serial RNG's. Since all numbers are generated by a serial RNG in a linear order, this order provides a natural measure of "distance" between adjacent random numbers, which Dieharder can use to look for correlations. When using an RNG for a parallel program, however, this notion of "distance" is more complicated, because calls to the RNG can execute in parallel. The results in Figure 6-4 use numbers generated in a serial execution of the (parallel) test program, which should maximize the correlation between adjacent random numbers due to similarities in the corresponding pedigrees. In principle, another execution order of the same program could generate random numbers in a different order and lead to different Dieharder test results.

As a practical matter, DotMix uses $r = 4$ mixing iterations to generate empirically high-quality random numbers. The difference in performance per call to DotMix with $r = 0$ and with $r = 4$ is less than 2 %, and thus DotMix can generate high-quality random numbers without sacrificing performance.

6.4 Other pedigree-based DPRNG's

This section investigates several other pedigree-based schemes for DPRNG's. Principal among these schemes is LCGMix, which uses a compression function based on linear congruential generators and the same mixing function as DotMix. We prove that the probability that LCGMix's compression function generates a collision is small, although not quite as small as for DotMix. We examine Dieharder results which indicate that LCGMix is statistically good. We also discuss alternative DPRNG schemes and their utility. These DPRNG's demonstrate that pedigrees can enable not only DotMix, but a host of other

DPRNG implementations. We close by observing a theoretical weakness with DotMix which would be remedied by a 4-independent compression scheme.

The LCGMix *DPRNG*

LCGMix is related to the "Lehmer tree" DPRNG scheme [139]. LCGMix uses a family of compression functions for pedigrees that generalize linear congruential generators (LCG's) [222, 243]. LCGMix then "RC6-mixes" the compressed pedigree to generate a pseudorandom value using the same mixing function as DotMix.

The basic idea behind LCGMix is to compress a pedigree by combining each successive rank using an LCG that operates modulo a prime p, where p is close to but less than $m = 2^w$, where w is the computer word width. LCGMix uses only three random nonzero values $\alpha, \beta, \gamma \in \mathbb{Z}_p$, rather than a table as DotMix does. Specifically, for an instruction x at depth $d = d(x)$ with pedigree $J(x) = \langle j_1, j_2, \ldots, j_d \rangle$, the LCGMix compression function performs the following recursive calculation modulo p:

$$X_d = \begin{cases} \gamma & \text{if } d = 0, \\ \alpha X_{d-1} + \beta j_d & \text{if } d > 0. \end{cases}$$

The value X_d is the compressed pedigree. Thus, the LCGMix compression function need only perform two multiplications modulo p and one addition modulo p per rank in the pedigree.

Assume, as for DotMix, that the spawn depth $d(x)$ for any instruction x in a dynamic multithreaded program is bounded by $d(x) \le D$. The family of compression functions used by LCGMix can be defined as follows:

Definition 2 *Let α, β, γ be nonzero integers chosen uniformly at random from $\mathbb{Z}_p$. Define $c_{\alpha,\beta,\gamma} : \bigcup_{d=1}^{\infty} \mathbb{Z}_p^d \to \mathbb{Z}_p$ by*

$$c_{\alpha,\beta,\gamma}(J) = \left(\alpha^d \gamma + \beta \sum_{k=1}^{d} \alpha^{d-k} j_k \right) \bmod p \ ,$$

where $J = \langle j_1, j_2, \ldots, j_d \rangle \in \mathbb{Z}_p^d$. The **LCGMix** ***compression-function family*** *is the set of functions*

$$C_{\text{LCGMix}} = \{ c_{\alpha,\beta,\gamma} : \alpha, \beta, \gamma \in \mathbb{Z}_p - \{0\} \} \,.$$

The next theorem shows that the probability a randomly chosen compression function $c_{\alpha,\beta,\gamma} \in C_{\text{LCGMix}}$ hashes two distinct pedigrees to the same value is small, although not quite as small as for DotMix.

Theorem 41 *Let $c_{\alpha,\beta,\gamma} \in C_{\text{LCGMix}}$ be a randomly chosen compression function. Then for any two distinct pedigrees $J \in \mathbb{Z}_p^d$ and $J' \in \mathbb{Z}_p^{d'}$ we have $\Pr\{c_{\alpha,\beta,\gamma}(J) = c_{\alpha,\beta,\gamma}(J')\} \le D/(p-1)$, where $D = \max\{d, d'\}$.*

Proof. Let $J = \langle j_1, j_2, \ldots, j_d \rangle$ and $J' = \left\langle j'_1, j'_2, \ldots, j'_{d'} \right\rangle$. The important observation is that the difference $c_{\alpha,\beta,\gamma}(J) - c_{\alpha,\beta,\gamma}(J')$ is a nonzero polynomial in α of degree at most D with coefficients in $\mathbb{Z}_p$. Thus, there are at most D roots to the equation $c_{\alpha,\beta,\gamma}(J) - c_{\alpha,\beta,\gamma}(J') = 0$, which are values for α that cause the two compressed pedigrees to collide. Since there are $p-1$ possible values for α, the probability of collision is at most $D/(p-1)$. $\square$

This pairwise-collision probability implies a theoretical bound on how many random numbers LCGMIX can generate before one would expect a collision between any pair of numbers in the set. By Boole's Inequality [100, p. 1195], compressing n pedigrees with a random function from C_{LCGMIX} gives a collision probability between any pair of those n compressed pedigrees of at most $\binom{n}{2}D/(p-1) = n(n-1)D/2(p-1)$. With $p = 2^{64} - 59$ and making the reasonable assumption that $D \leq 100$, the probability that compressing 500,000 pedigrees results in a collision is less than 1 in a million. As can be seen, 500,000 pedigrees is a factor of 10 less than the 5 million for DOTMIX for the same probability. Since our implementation of LCGMIX was no faster than our implementation of DOTMIX per function call, we favored the stronger theoretical guarantee of DOTMIX for the Cilk Plus library.

We tested the quality of random numbers produced by LCGMIX using Dieharder, producing the results in Figure 6-4. The data suggest that, as with DOTMIX, $r = 4$ mixing iterations in LCGMIX are sufficient to provide random numbers whose statistical quality is comparable to those produced by the Mersenne twister.

Further ideas for DPRNG's

We can define DPRNG's using families of compression functions that exhibit stronger theoretical properties or provide faster performance than either DOTMIX or LCGMIX.

One alternative is to use ***tabulation hashing*** [80] to compress pedigrees, giving compressed pedigree values that are 3-independent and have other strong theoretical properties [311]. This DPRNG is potentially useful for applications that require stronger properties from their random numbers. To implement this scheme, the compression function treats the pedigree as a bit vector whose 1 bits select entries in a table of random values to XOR together.

As another example which favors theoretical quality over performance, a DPRNG could be based on compressing the pedigree with a SHA-1 hash [294], providing a cryptographically secure compression function that would not require any mixing to generate high-quality random numbers. Other cryptographically secure hash functions could be used as well. While cryptographically secure hash functions are typically slow, they would allow the DPRNG to provide pseudorandom numbers with very strong theoretical properties, which can be important for some applications.

On the other side of the performance-quality spectrum, a DPRNG could be based on compressing a pedigree using a faster hash function. One such function is the hash function used in UMAC [48], which performs half the multiplications of DOTMIX's compression function. The performance of the UMAC compression scheme and the quality of the DPRNG it engenders offers an interesting topic for future research.

4-independent compression of pedigrees

Although Theorem 40 shows that the probability is small that DOTMIX's compression function causes two pedigrees to collide, DOTMIX contains a theoretical weakness. Consider two distinct pedigrees J_1 and J_2 of length D, and suppose that DOTMIX maps J_1 and J_2 to the same value, or more formally, that DOTMIX chooses a compression function c_Γ such that $c_\Gamma(J_1) = c_\Gamma(J_2)$. Let $J + \langle j \rangle$ denote the pedigree that results from appending the rank j to

```
17 template <typename T>
18 class DPRNG {
19   DPRNG();                     // Constructor
20   ~DPRNG();                    // Destructor
21   DPRNG_scope current_scope(); // Get current scope
22   void set(uint64_t seed, DPRNG_scope scope); // Init
23   uint64_t get();              // Get random number
24 };
```

Figure 6-5: A C++ interface for a pedigree-based DPRNG suitable for use with Cilk Plus. The type `T` of the DPRNG object specifies a particular DPRNG library, such as DotMix, that implements this interface. In addition to accepting an argument for an initial seed, the initialization method for the DPRNG in line 22 also requires an lexical scope, restricting the scope where the DPRNG object can be used.

the pedigree J. Because J_1 and J_2 both have length D, it follows that

$$\begin{aligned} c_\Gamma(J_1 + \langle j \rangle) &= c_\Gamma(J_1) + \gamma_{D+1} j \\ &= c_\Gamma(J_2) + \gamma_{D+1} j \\ &= c_\Gamma(J_2 + \langle j \rangle) \ . \end{aligned}$$

Thus, DotMix hashes the pedigrees $J_1 + \langle j \rangle$ and $J_2 + \langle j \rangle$ to the same value, regardless of the value of j. In other words, one collision in the compression of the pedigrees for two strands, however rare, might result in many ancillary collisions.

To address this theoretical weakness, a DPRNG scheme might provide the guarantee that if two pedigrees for two strands collide, then the probability remains small that the pedigrees collide for any other pair of strands. A 4-independent hash function [402] would achieve this goal by guaranteeing that the probability is small that any sequence of 4 distinct pedigrees hash to any particular sequence of 4 values. Tabulation-based 4-independent hash functions for single words are known [389], but extending these techniques to hash pedigrees efficiently remains an intriguing open problem.

6.5 A scoped DPRNG library interface

This section presents the programming interface for a DPRNG library that we implemented for Cilk Plus. This interface demonstrates how programmers can use a pedigree-based DPRNG library in applications. The interface uses the notion of "scoped" pedigrees, which allow DPRNG's to compose easily.

Scoped pedigrees solve the following problem. Suppose that a dynamic multithreaded program contains a parallel subcomputation that uses a DPRNG, and suppose that the program would like to run this subcomputation the same way twice. Using scoped pedigrees, the program can guarantee that both runs generate the exact same random numbers, even though corresponding RNG calls in the subcomputations have different pedigrees globally.

Programming interface

Figure 6-5 shows a C++ interface for a DPRNG suitable for use with Cilk Plus. It resembles the interface for an ordinary serial RNG, but it constrains when the DPRNG can be used to generate random numbers by defining a "scope" for each DPRNG instance. The `set` method in line 22 initializes the DPRNG object based on two quantities: an initial seed and

```
25 uint64_t foo(DPRNG<DotMix>* rand, uint64_t seed, int i) {
26   uint64_t sum = 0;
27   DPRNG_scope scope = rand->current_scope();
28   rand->set(seed, scope);
29   for (int j = 0; j < 15; ++j) {
30     uint64_t val = rand->get();
31     sum += val;
32   }
33   return sum;
34 }
35
36 int main(void) {
37   const int NSTREAMS = 10;
38   uint64_t sum[NSTREAMS];
39   uint64_t s1 = 0x42;  uint64_t s2 = 31415;
40   // Generate NSTREAMS identical streams
41   cilk_for (int i = 0; i < NSTREAMS; ++i) {
42     DPRNG<DotMix>* rand = new DPRNG<DotMix>();
43     sum[i] = foo(rand, s1, i);
44     sum[i] += foo(rand, s2, i);
45     delete rand;
46   }
47   for (int i = 1; i < NSTREAMS; ++i)
48     assert(sum[i] == sum[0]);
49   return 0;
50 }
```

Figure 6-6: A program that generates `NSTREAMS` identical streams of random numbers. Inside function `foo`, the code in lines 27–28 limits the scope of `rand` so that it can generate random numbers only within `foo`.

a scope. The seed is the same as for an ordinary serial RNG. The ***scope*** represented by a pedigree J is the set of instructions whose pedigrees have J as a common prefix. Specifying a scope (represented by) J to the DPRNG object `rand` restricts the DPRNG to generate numbers only within that scope and to ignore the common prefix J when generating random numbers. By default, the programmer can pass in the global scope $\langle 0 \rangle$ to let the DPRNG object be usable anywhere in the program. The interface allows programmers to limit the scope of a DPRNG object by getting an explicit scope (line 21) and setting the scope of a DPRNG object (line 22).

Figure 6-6 demonstrates how restricting the scope of a DPRNG can be used to generate repeatable streams of random numbers within a single program. Inside `foo`, the code in lines 27–28 limits the scope of `rand` so that it generates random numbers only within `foo`. Because of this limited scope, the assertion in line 48 holds true. If the programmer sets `rand` with a global scope, then each call to `foo` would generate a different value for `sum`, and the assertion would fail.

Intuitively, one can think of the scope as extension of the seed for a serial RNG. To generate exactly the same stream of random numbers in a dthreaded program, one must (1) use the same seed, (2) use the same scope, and (3) have exactly the same structure of spawned functions and RNG calls within the scope. Even if `foo` from Figure 6-6 were modified to generate random numbers in parallel, the use of scoped pedigrees still guarantees that each iteration of the parallel loop in line 41 behaves identically.

Implementation

To implement the `get` method of a DPRNG, we use the API in Figure 6-2 to extract the current pedigree during the call to `get`, and then we hash the pedigree. The principal remaining difficulty in the DPRNG's implementation is in handling scopes.

Intuitively, a scope can be represented by a pedigree prefix that should be common to the pedigrees of all strands generating random numbers within the scope. Let y be the instruction corresponding to a call to `current_scope`(), and let x be a call to `get`() within the scope $J(y)$. Let $J(x) = \langle j_1, j_2, \ldots, j_{d(x)} \rangle$ and $J(y) = \left\langle j'_1, j'_2, \ldots, j'_{d(y)} \right\rangle$. Since x belongs to the scope $J(y)$, it follows that $d(x) \geq d(y)$, and we have $j_{d(y)} \geq j'_{d(y)}$ and $j_k = j'_k$ for all $k < d(y)$. We now define the ***scoped pedigree*** of x with respect to scope $J(y)$ as

$$J_{J(y)}(x) = \left\langle j_{d(y)} - j'_{d(y)}, j_{d(y)+1}, \ldots, j_{d(x)} \right\rangle .$$

To compute a random number for $J(x)$ excluding the scope $J(y)$, we simply perform a DPRNG scheme on the scoped pedigree $J_{J(y)}(x)$. For example, DotMix computes $\mu(c_\Gamma(J_{J(y)}(x)))$. Furthermore, one can check for scoping errors by verifying that $J(y)$ is indeed the prefix of $J(x)$ via a direct comparison of all the pedigree terms.

Scoped pedigrees allow DPRNG's to optimize the process of reading a pedigree. By hashing scoped pedigrees, a call to the DPRNG need only read a suffix of the pedigree, i.e. the scoped pedigree itself, rather than the entire pedigree. To implement this optimization, each scope may store a pointer to the spawn parent for the deepest rank of the scope, and then the code for reading the pedigree extracts ranks as usual until it observes the spawn parent the scope points to. One problem with this optimization is that a DPRNG may not detect if it is hashing a pedigree outside of its scope. To overcome this problem, DotMix supports a separate mode for debugging, in which each call checks its pedigree term-by-term to verify that it is within the scope.

6.6 Performance results

This section reports on our experimental results investigating the overhead of maintaining pedigrees and the cost of the DotMix DPRNG. To study the overhead of tracking pedigrees, we modified the open-source MIT Cilk [146], whose compiler and runtime system were both accessible. We discovered that the overhead of tracking pedigrees is small, having a geometric mean of only 1 % on all tested benchmarks. To measure the costs of DotMix, we implemented it as a library for a version of Cilk Plus that Intel engineers had augmented with pedigree support, and we compared its performance to a nondeterministic DPRNG implemented using worker-local Mersenne twister RNG's. Although the price of determinism from using DotMix was approximately a factor of 2.3 greater per function call than Mersenne twister on a synthetic benchmark, this price was much smaller on more realistic codes such as a sample sort and Monte Carlo simulations. These empirical results suggest that pedigree-based DPRNG's are amply fast for debugging purposes and that their overheads may be low enough for some production codes.

Pedigree overheads

To estimate the overhead of maintaining pedigrees, we ran a set of microbenchmarks for MIT Cilk with and without support for pedigrees. We modified MIT Cilk 5.4.6 to store the

Application	*Default*	*Pedigree*	*Overhead*
`fib`	11.03	12.13	1.10
`cholesky`	2.75	2.92	1.06
`fft`	1.51	1.53	1.01
`matmul`	2.84	2.87	1.01
`rectmul`	6.20	6.21	1.00
`strassen`	5.23	5.24	1.00
`queens`	4.61	4.60	1.00
`plu`	7.32	7.35	1.00
`heat`	2.51	2.46	0.98
`lu`	7.88	7.25	0.92

Figure 6-7: Overhead of maintaining 64-bit rank pedigree values for the Cilk benchmarks as compared to the default of MIT Cilk 5.4.6. The experiments were run on an AMD Opteron 6168 system with a single 12-core CPU clocked at 1.9 GHz. All times are the minimum of 15 runs measured in seconds.

necessary 64-bit rank values and pointers in each frame for spawn pedigrees and to maintain spawn pedigrees at runtime. We then ran 10 MIT Cilk benchmark programs using both our modified version of MIT Cilk and the original MIT Cilk. In particular, we ran the following benchmarks:

- `fib`: Recursive exponential-time calculation of the 40th Fibonacci number.
- `cholesky`: A divide-and-conquer Cholesky factorization of a sparse 2000×2000 matrix with 10,000 nonzeros.
- `fft`: Fast Fourier transform on 2^{22} elements.
- `matmul`: Recursive matrix multiplication of 1000×1000 square matrices.
- `rectmul`: Rectangular matrix multiplication of 2048×2048 square matrices.
- `strassen`: Strassen's algorithm for matrix multiplication on 2048×2048 square matrices.
- `queens`: Backtracking search to count the number of solutions to the 24-queens puzzle.
- `plu`: LU-decomposition with partial pivoting on a 2048×2048 matrix.
- `heat`: Jacobi-type stencil computation on a 4096×1024 grid for 100 timesteps.
- `lu`: LU-decomposition on a 2048×2048 matrix.

The results from these benchmarks, as summarized in Figure 6-7, show that the slowdown due to spawn pedigrees is generally negligible, having a geometric mean of less than 1 %. Although the overheads run as high as 10 % for `fib`, they appear to be within measurement noise caused by the intricacies of modern-day processors. For example, two benchmarks actually run measurably faster despite the additional overhead. This benchmark suite gives us confidence that the overhead for maintaining spawn pedigrees should be close to negligible for most real applications.

DPRNG overheads

To estimate the cost of using DPRNG's, we persuaded Intel to modify its Cilk Plus concurrency platform to maintain pedigrees, and then we implemented the DotMix DPRNG for Intel Cilk Plus.[6] We compared DotMix's performance, using $r = 4$ mixing iterations, to a nondeterministic parallel implementation of the Mersenne twister on synthetic benchmarks, as well as on more realistic applications. From these results, we estimate that the "price of

[6]The Intel C/C++ compiler v12.1 provides compiler and runtime support for maintaining pedigrees in Cilk.

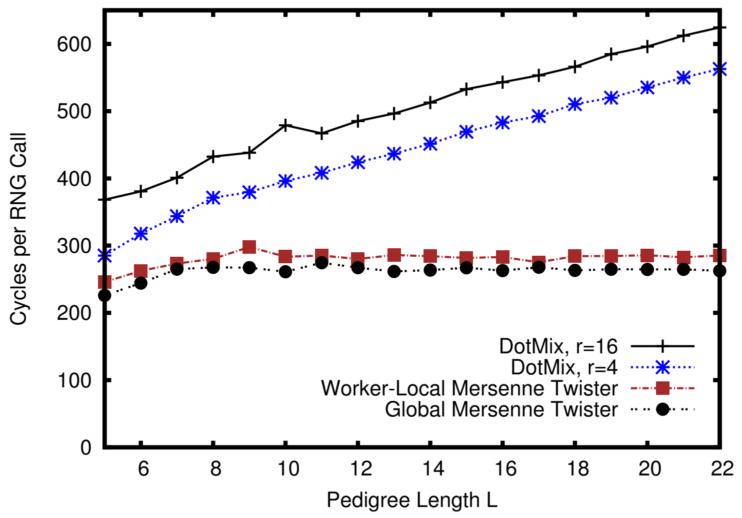


Figure 6-8: Overhead of various RNG's on the CBT benchmark when generating $n = 2^{20}$ random numbers. Each data point represents the minimum of 20 runs. The global Mersenne twister RNG from the GSL library [147] only works for serial code, while the worker-local Mersenne twister is a nondeterministic parallel implementation.

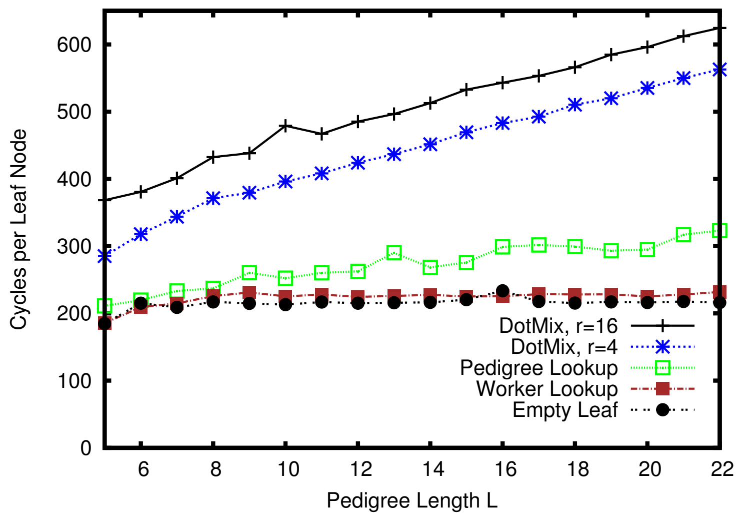


Figure 6-9: Breakdown of overheads of DotMix in the CBT benchmark, with $n = 2^{20}$. This experiment uses the same methodology as for Figure 6-8.

determinism" for DotMix is about a factor of 2.3 in practice on synthetic benchmarks that generate large pedigrees, but it can be significantly less for more practical applications. For these experiments, we coded by hand an optimization that the compiler could, but does not currently, implement. To avoid incurring the overhead of multiple worker lookups on every call to generate a random number, within a strand, DotMix looks up the worker once and uses it for all calls to the API made by the strand.

We used Intel Cilk Plus to perform three different experiments. First, we used a synthetic benchmark, called CBT, to quantify how DotMix performance is affected by pedigree length. Next, we used CBT to measure the performance benefits of flattening pedigrees for `cilk_for` loops. Finally, we benchmarked the performance of DotMix on realistic applications that require random numbers. All experiments described in the remainder of

CPU	Intel Xeon X5650
Clock	2.67 GHz
Cores per processor chip	6
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	128 KiB
L3 cache/socket	12 MiB
DRAM	49 GiB
Compiler	Intel C/C++ compiler v13.0 beta

Figure 6-10: Technical specifications of the machine used for benchmarking.

this section were run on the Intel Xeon machine described in Figure 6-10. The code was compiled using the Intel C/C++ compiler with `-O3` optimizations and uses the Intel Cilk Plus runtime, which together provide support for pedigrees.

The CBT benchmark that we constructed successively creates n/k complete binary trees, each with k leaves, which it walks in parallel by spawning two children recursively. The pedigree of each leaf has uniform length $L = 2 + \lg k$, and within each leaf the RNG is called. Figure 6-8 compares the performance of various RNG's on the CBT benchmark, fixing $n = 2^{20}$ random numbers but varying the pedigree length L. These results show that the overhead of DotMix increases roughly linearly with pedigree length, but that DotMix is still within about a factor of 2 compared to using a Mersenne Twister RNG. From a linear regression on the data from Figure 6-8, we observed that the cost per additional term in the pedigree for both DotMix and LCGMix was about 15 cycles, regardless of whether $r = 4$ or $r = 16$.[7]

Figure 6-9 breaks down the overheads of DotMix in the CBT benchmark further. To generate a random number, DotMix requires looking up the currently executing worker in Cilk (from thread-local storage),[8] reading the pedigree, and then generating a random number. The figure compares the overhead of DotMix with the overhead of simply spawning a binary tree with n leaves while performing no computation within each leaf. From these data, we can attribute at least 35 % of the execution time of DotMix calls to the overhead of simply spawning the tree itself. Furthermore, by measuring the cost of reading a pedigree, we observe that for the longest pedigrees, roughly half of the cost of an RNG call can be attributed to looking up the pedigree itself.

Pedigree flattening

To estimate the performance improvement of the pedigree-flattening optimization for parallel loops described in Section 6.2, we compared the cost performing n pedigree lookups for the CBT benchmark (Figure 6-9) to the cost of pedigree lookups in a `cilk_for` loop performing n pedigree lookups in parallel. Figure 6-11 shows that the `cilk_for` pedigree optimization substantially reduces the cost of pedigree lookups. This result is not surprising, since the pedigree lookup for recursive spawning in the CBT benchmark cost increases roughly linearly with $\lg n$, whereas the lookup cost remains nearly constant for using a `cilk_for` as $\lg n$ increases.

[7]This linear model overestimates the running time of this benchmark for $L < 4$ (not shown). For small trees, it is difficult to accurately measure and isolate the RNG performance from the cost of the recursion itself.

[8]Our implementation of a parallel RNG based on Mersenne Twister also requires a similar lookup from thread-local storage to find the worker's local RNG.

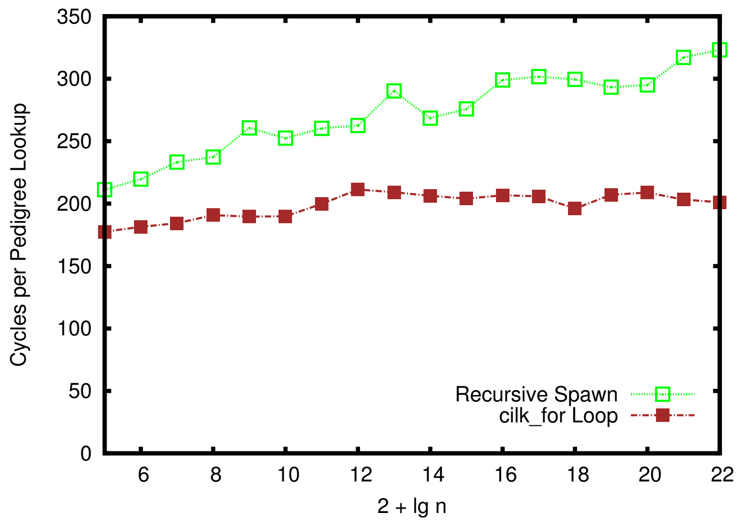


Figure 6-11: Comparison of pedigree lookups in a `cilk_for` loop with recursive spawns in a binary tree. The recursive spawning generates n leaves as in the CBT benchmark, with each pedigree lookup having $L = 2 + \lg n$ terms. The `cilk_for` loop uses a grain size of 1, and generates pedigrees of length 4.

Application	T_1(`DotMix`)/T_1(`mt`)	T_{12}(`DotMix`)/T_{12}(`mt`)
`fib`	2.33	2.25
`pi`	1.21	1.13
`maxIndSet`	1.14	1.08
`sampleSort`	1.00	1.00
`DiscreteHedging`	1.03	1.03

Figure 6-12: Overhead of DotMix as compared to a parallel version of the Mersenne twister (denoted by `mt` in the table) on four programs. All benchmarks use the same worker-local Mersenne twister RNG's as in Figure 6-8 except for `DiscreteHedging`, which uses QuantLib's existing Mersenne twister implementation.

Application benchmarks

Figure 6-12 summarizes the performance results for the various RNG's on four application benchmarks:

- `pi`: A simple Monte-Carlo simulation that calculates the value of the transcendental number π using 256 M samples.
- `maxIndSet`: A randomized algorithm for finding a maximal independent set in graphs with approximately 16 M vertices, where nodes have an average degree of between 4 and 20.
- `sampleSort`: A randomized recursive samplesort algorithm on 64 M elements, with the base case on 10,000 samples.
- `DiscreteHedging`: A financial-model simulation using Monte Carlo methods.

We implemented the `pi` benchmark ourselves. The `maxIndSet` and `sampleSort` benchmarks were derived from the code described in [52]. The `DiscreteHedging` benchmark is derived from the QuantLib library for computation finance. More specifically, we modified QuantLib version 1.1 to parallelize this example as described in [179], and then supplemented QuantLib's existing RNG implementation of Mersenne Twister with DotMix.

To estimate the per-function-call cost of DotMix, we also ran the same `fib` benchmark that was used for the experiment described in Figure 6-7, but modified so that the RNG is called once at every node of the computation. The results for `fib` in Figure 6-12 indicate that DotMix is about a factor of 2.3 slower than using Mersenne twister, suggesting that the price of determinism for parallel random-number generation in dynamic multithreaded programs is at most 2–3 per function call.

The remaining applications pay a relatively lesser price for determinism for two reasons. First, many of these applications perform more computation per random number obtained, thereby reducing the relative cost of each call to DotMix. Second, many of these applications call DotMix within a `cilk_for` loop, and thus benefit from the pedigree-flattening optimization to reduce the cost per call of reading the pedigree.

6.7 Related work

The problem of generating random numbers deterministically in multithreaded programs has received significant attention. SPRNG [273] is a popular DPRNG for Pthreading platforms that works by creating independent RNG's via a parameterization process. Other approaches to parallelizing RNG's exist, such as leapfrogging and splitting. Coddington [92] surveys these alternative schemes and their respective advantages and drawbacks. It may be possible to adapt some of these Pthreading RNG schemes to create similar DPRNG's for dynamic multithreaded programs.

The concept of deterministically hashing interesting locations in a program execution is not new. The `maxIndSet` and `sampleSort` benchmarks we borrowed from [52] used an *ad hoc* hashing scheme to afford repeatability, a technique we have used technique ourselves in the past and which must have been reinvented numerous times before us. More interesting is the pedigree-like scheme due to Bond and McKinley [62] where they use an LCG strategy similar to LCGMix to assign deterministic identifiers to calling contexts for the purposes of residual testing, anomaly-based bug detection, and security intrusion detection.

Salmon *et al.* [338] independently explored the idea of "counter-based" parallel RNG's, which generate random numbers by performing independent transformations of counter values. Counter-based RNG's use similar ideas to pedigree-based DPRNG's. Intuitively, the compressed pedigree values generated by DotMix and LCGMix can be thought of as counter values, and the mixing function corresponds to a particular kind of transformation. Salmon *et al.* focus on generating high-quality random numbers, exploring several transformations based on both existing cryptographic standards and some new techniques, and show that these transformations lead to RNG's with good statistical properties. Counter-based RNG's do not directly lead to DPRNG's for a dynamic multithreaded programs, however, because it can be difficult to generate deterministic counter values. One can, however, apply these transformations to compressed pedigree values and automatically derive additional pedigree-based DPRNG's.

6.8 Concluding remarks

DotMix supports a simple programming model for generating pseudorandom numbers deterministically in parallel. DotMix employs the pedigree mechanism, which leverages the simple programming model of Cilk programs to quickly assign uniquely identifiers to all program points in a deterministic, processor-oblivious fashion. We conclude by discussing two enhancements for pedigrees and DPRNG's. We also consider how the notion of pedigrees might be extended to work on other concurrency platforms.

The first enhancement addresses the problem of multiple calls to a DPRNG within a strand. The mechanism described in Section 6.2 involves calling the StrandBreak function, which increments the rank whenever a call to the DPRNG is made, thereby ensuring that two successive calls have different pedigrees. An alternative idea is to have the DPRNG store for each worker p an ***event counter*** e_p that the DPRNG updates manually and uses as an additional pedigree term so that multiple calls to the DPRNG per strand generate different random numbers.

The DPRNG can maintain an event counter for each worker as follows. Suppose that the DPRNG stores for each worker p the last pedigree p read. When worker p calls the DPRNG to generate another random number, causing the DPRNG to read the pedigree, the DPRNG can check whether the current pedigree matches the last pedigree p read. If it matches, then p has called the DPRNG again from the same strand, and so the DPRNG updates e_p. If it does not match, then p must be calling the DPRNG from a new strand. Because each strand is executed by exactly one worker, the DPRNG can safely reset e_p to a default value in order to generate the next random number.

This event counter scheme improves the composability of DPRNG's in a program, because calls to one DPRNG do not affect calls to another DPRNG in the same program, as they do for the scheme from Section 6.2. In practice, however, event counters can hurt the performance of a DPRNG. From experiments with the `fib` benchmark, we found that an event-counter scheme runs approximately 20 %–40 % slower per function call than the scheme from Section 6.2, and thus we favored the use of StrandBreak for our main results. Nevertheless, more efficient ways to implement an event-counter mechanism might exist, which would enhance composability.

Our second enhancement addresses the problem of "climbing the tree" to access all ranks in the pedigree for each call to the DPRNG, the cost of which is proportional to the spawn depth d. Some compression functions, including Definitions 1 and 2, can be computed ***incrementally***, and thus results can be "memoized" to avoid walking up the entire tree to compress the pedigree. In principle, one could memoize these results in a ***frame-data*** cache — a worker-local cache of intermediate results — and then, for some computations, generate random numbers in $O(1)$ time instead of $O(d)$ time. Preliminary experiments with using frame-data caches indicate, however, that in practice, the cost of memoizing the intermediate results in every stack frame outweighs the benefits from memoization, even in an example such as `fib`, where the spawn depth can be quite large. Hence, we opted not to use frame-data caches for DotMix. Nevertheless, it is an interesting open question whether another memoization technique, such as selective memoization specified by the programmer, might improve performance for some applications.

We now turn to the question of how to extend the pedigree ideas to "less structured" dynamic multithreading concurrency platforms. For some parallel-programming models with less structure than Cilk, it might not be important to worry about DPRNG's at all, because these models do not encapsulate the nondeterminism of the scheduler. Thus, a DPRNG would seem to offer little benefit over a nondeterministic parallel RNG. Nevertheless, some models that support more complex parallel control than the fork-join model of Cilk do admit the writing of deterministic programs, and for these models, the ideas of pedigrees can be adapted.

As an example, Intel Threading Building Blocks [277,330] supports pipeline parallelism, in which each stage of the pipeline is a fork-join computation. For this control construct, one could maintain an outer-level pedigree to identify the stage in the pipeline and combine it with a pedigree for the interior fork-join computation within a stage.

Although Cilk programs produce instruction traces corresponding to fork-join graphs, the pedigree idea also seems to extend to general dags, at least in theory. One can define pedigrees on general dags as long as the children (successors) of a node are ordered. The rank of a node x indicates the birth order of x with respect to its siblings. Thus, a given pedigree (sequence of ranks) defines a unique path from the source of the task graph. The complication arises because in a general dag, multiple pedigrees (paths) can lead to the same node. Assuming there exists a deterministic procedure for choosing a particular path as the "canonical" pedigree, one can still base a DPRNG on canonical pedigrees. It remains an open question, however, as to how efficiently one can maintain canonical pedigrees in this more general case, which will depend on the particular parallel-programming model.

6.9 Recent developments

This work has been adopted into industry in various ways since it was conducted. Pedigrees were incorporated into Intel Cilk Plus and both the Intel [198] and GNU C/C++ compilers [369]. Intel also adopted DotMix algorithm for its deterministic parallel random-number generation library [378]. DotMix also provided the basis for the efficient splittable random-number generator [371] introduced in Java JDK8 by Steele, Lea, and Flood.

Chapter 7

On-the-Fly Pipeline Parallelism

This chapter presents the Cilk-P concurrency platform [239], a dynamic multithreading concurrency platform that supports pipeline parallelism. This work was conducted in collaboration with I-Ting Angelina Lee, Charles E. Leiserson, Jim Sukha, and Zhunping Zhang.

7.1 Introduction

Pipeline parallelism[1] [55,157,169,268,277,295,317,328,339,381] is a well-known parallel-programming pattern that can be used to parallelize a variety of applications, including streaming applications from the domains of video, audio, and digital signal processing. Many applications, including the *ferret*, *dedup*, and *x264* benchmarks from the PARSEC benchmark suite [45,46], exhibit parallelism in the form of a ***linear*** pipeline, where a linear sequence $\mathcal{S} = \langle \mathcal{S}_0, \ldots, \mathcal{S}_{m-1} \rangle$ of abstract functions, called ***stages***, are executed on an input stream $I = \langle a_0, a_1, \ldots, a_{n-1} \rangle$. Conceptually, a linear pipeline can be thought of as a loop over the elements of I, where each loop ***iteration*** i processes an element a_i of the input stream. The loop body encodes the sequence $\mathcal{S}$ of stages through which each element is processed. Parallelism arises in linear pipelines because the execution of iterations can overlap in time, that is, iteration i may start after the preceding iteration $i-1$ has started, but before $i-1$ has necessarily completed.

Most systems that provide pipeline parallelism employ a ***construct-and-run*** model, as exemplified by the pipeline model in Intel Threading Building Blocks (TBB) [277], where the pipeline stages and their dependencies are defined *a priori* before execution. Systems that support construct-and-run pipeline parallelism are described in: [6,97,169,268,271,277, 305,307,317,327,339,381,388].

We have extended the Cilk parallel-programming model [146,196,246] to Cilk-P, a system that augments Cilk's native fork-join parallelism with ***on-the-fly*** pipeline parallelism, where the linear pipeline is constructed dynamically as the program executes. The Cilk-P system provides a flexible processor-oblivious linguistic model for pipelining that allows the structure of the pipeline to be determined dynamically as a function of data in the input stream. For example, Cilk-P allows pipelines to have a variable number of stages across iterations. The Cilk-P programming model is flexible, yet restrictive enough to allow provably efficient scheduling, as Sections 7.5 through 7.8 will show. In particular, Cilk-P's scheduler provides automatic "throttling" to ensure that the computation uses bounded space. As a testament

[1]Pipeline parallelism should not be confused with instruction pipelining in hardware [333] or software pipelining [229].

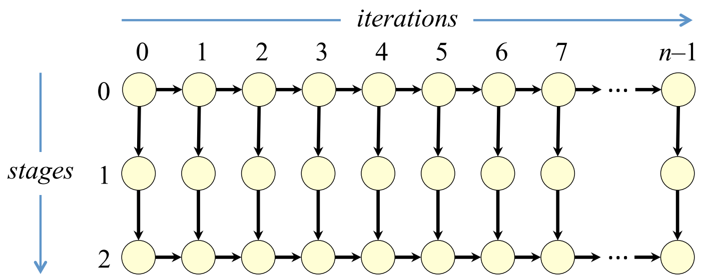


Figure 7-1: Modeling the execution of *ferret*'s linear pipeline as a pipeline dag. Each column contains nodes for a single iteration, and each row corresponds to a stage of the pipeline. Vertices in the dag correspond to nodes of the linear pipeline, and edges denote dependencies between the nodes. Throttling edges are not shown.

to the flexibility provided by Cilk-P, we were able to parallelize the *x264* benchmark from PARSEC, an application that cannot be programmed easily using TBB [328]. These flexible linguistics and provable performance guarantees simplify the task of engineering efficient multicore programs that exhibit a mixture of fork-join and pipeline parallelism.

Although Cilk-P's support for defining linear pipelines on the fly is more flexible than construct-and-run approaches and the `ordered` directive in OpenMP [305], which supports a limited form of on-the-fly pipelining, it is less expressive than other approaches. Blelloch and Reid-Miller [55] describe a scheme for on-the-fly pipeline parallelism that employs futures [30, 141] to coordinate the stages of the pipeline, allowing even nonlinear pipelines to be defined on the fly. Although futures permit more complex, nonlinear pipelines to be expressed, this generality can lead to unbounded space requirements to attain even modest speedups [57].

To illustrate the ideas behind the Cilk-P model, consider a simple 3-stage linear pipeline such as in the *ferret* benchmark from PARSEC [45,46]. Figure 7-1 shows a ***pipeline dag*** $G = (V, E)$ representing the execution of the pipeline. Each of the 3 horizontal rows corresponds to a stage of the pipeline, and each of the n vertical columns is an iteration. A vertex in the dag represents a pipeline ***node*** $(i, j) \in V$, where $i = 0, 1, \ldots, n-1$ and $j = 0, 1, 2$, which corresponds to the execution of $\mathcal{S}_j(a_i)$, the jth stage in the ith iteration. The edges between nodes denote control dependencies. A ***stage edge*** from node (i, j) to node (i, j'), where $j < j'$, indicates that (i, j') cannot start until (i, j) completes. A ***cross edge*** from node $(i-1, j)$ to node (i, j) indicates that (i, j) can start execution only after node $(i-1, j)$ completes. Cilk-P always executes nodes of the same iteration in increasing order by stage number, thereby creating a vertical chain of stage edges. Cross edges between corresponding stages of adjacent iterations are optional.

We can categorize the stages of a Cilk-P pipeline. A stage is a ***serial stage*** if all nodes belonging to the stage are connected by cross edges. A stage is a ***parallel stage*** if none of the nodes belonging to the stage are connected by cross edges. A stage is a ***hybrid stage*** if it is neither a serial nor a parallel stage. For example, the *ferret* pipeline, illustrated in Figure 7-1, exhibits a static structure often referred to as an "SPS" pipeline, since stage 0 and stage 2 are serial and stage 1 is parallel. Cilk-P requires that pipelines be linear, meaning that stages within an iteration must be executed one after another. Stage 0 of any Cilk-P pipeline is always a serial stage. Later stages may be serial, parallel, or hybrid, as we shall see in Sections 7.2 and 7.3.

To execute a linear pipeline, Cilk-P follows the lead of TBB and adopts a ***bind-to-element*** approach [268,277], where workers execute pipeline iterations either to completion or until an unresolved dependency is encountered. In particular, Cilk-P and TBB both rely on work-stealing schedulers for load balancing. In contrast, many systems that support pipeline parallelism, including typical Pthreaded implementations, execute linear pipelines using a ***bind-to-stage*** approach, where each worker executes a distinct stage and coordination between workers is handled using concurrent queues [169, 339, 388]. Some researchers report that the bind-to-element approach generally outperforms bind-to-stage [295, 328], since a work-stealing scheduler can do a better job of dynamically load-balancing the computation, but our own experiments show mixed results.

A natural theoretical question is, how much parallelism is inherent in the *ferret* pipeline (or in any pipeline)? How much speedup can one hope for? Since the computation is represented as a dag $G = (V, E)$, one can use a simple work-span analysis to answer this question. In this analytical model, we assume that each vertex $v \in V$ executes in some time $w(v)$. For simplicity, let us assume that the parallel pipeline is deterministic. The work T_1 of the computation is $T_1 = \sum_{v \in V} w(v)$. The span T_∞ of the computation is the length of a longest weighted path through G, which is essentially the time of an infinite-processor execution. The parallelism T_1/T_∞ captures the maximum possible speedup attainable on any number of processors, using any scheduler.

We can apply the work-span analysis to the *ferret* pipeline shown in Figure 7-1. This pipeline has the special structure that each node executes serially, that is, without any nested parallelism inside the node. Thus, in the *ferret* pipeline, the work and span of each node is the same. Let $w(i, j)$ be the execution time of node (i, j). Assume that the serial stages 0 and 2 execute in unit time — for all i, we have $w(i, 0) = w(i, 2) = 1$ — and that the parallel stage 1 executes in time $r \gg 1$, that is, for all i, we have $w(i, 1) = r$. The work of this pipeline is therefore $T_1 = n(r + 2)$. Because the pipeline dag is grid-like, the span of this SPS pipeline can be realized by some staircase walk through the dag from node $(0, 0)$ to node $(n - 1, 2)$, and therefore the span is

$$\begin{aligned} T_\infty &= \max_{0 \le x < n} \left\{ \sum_{i=0}^{x} w(i,0) + w(x,1) + \sum_{i=x}^{n-1} w(i,2) \right\} \\ &= n + r \ . \end{aligned}$$

Consequently, the parallelism of this dag is $T_1/T_\infty = n(r + 2)/(n + r)$, which for $1 \ll r \le n$ is at least $r/2 + 1$. Thus, if stage 1 contains much more work than the other two stages, the *ferret* pipeline exhibits good parallelism.

On an ideal shared-memory computer, Cilk-P guarantees to execute the *ferret* pipeline efficiently. In particular, Cilk-P guarantees linear speedup on a computer with up to $T_1/T_\infty = O(r)$ processors. Generally, Cilk-P executes a pipeline with linear speedup as long as the parallelism of the pipeline exceeds the number of processors on which the computation is scheduled. Moreover, as Section 7.3 will describe, Cilk-P allows stages of the pipeline themselves to be parallel using recursive pipelining or fork-join parallelism.

In practice, it is also important to limit the space used during an execution. Unbounded space can cause thrashing of the memory system, leading to slowdowns not predicted by simple execution models. In particular, a bind-to-element scheduler must avoid creating a ***runaway*** pipeline — a situation where the scheduler allows many new iterations to be started before finishing old ones. In Figure 7-1, a runaway pipeline might correspond to executing many nodes in stage 0 (the top row) without finishing the other stages of the computation in the earlier iterations. Runaway pipelines can cause space utilization to grow unboundedly, because every started but incomplete iteration requires space to store local variables.

Cilk-P automatically ***throttles*** pipelines to avoid runaway pipelines. On a system with P workers, Cilk-P inhibits the start of iteration $i+K$ until iteration i has completed, where $K = \Theta(P)$ is the ***throttling limit***. In terms of the pipeline dag, throttling corresponds to putting ***throttling edges*** from the last node in each iteration i to the first node in iteration $i+K$. For the simple pipeline from Figure 7-1, throttling does not adversely affect asymptotic scalability if stages are uniform, but it can be a concern for more complex pipelines, as Section 7.11 will discuss. The Cilk-P scheduler guarantees efficient scheduling of pipelines as a function of the parallelism of the dag in which throttling edges are included in the calculation of span.

Cilk-P supports a simple, principled approach to writing programs that exhibit pipeline parallelism and to reasoning about their performance. Cilk-P provides a simple, yet flexible, programming model for writing fast, deterministic programs that exhibit on-the-fly pipeline parallelism. As in the Cilk model, Cilk-P's linguistics encapsulate the nondeterminism of synchronization mechanisms for enforcing pipeline dependencies and offer a flexible, processor-oblivious model for writing programs with pipeline parallelism. These properties can assist programmers in reasoning about the correctness of these programs. Furthermore, the Cilk-P model admits work-span analysis for understanding the scalability of pipeline programs. Cilk-P's provably efficient scheduling guarantees to execute a Cilk-P program near optimally on whatever processors are available at runtime.

Contributions

Our prototype Cilk-P system adapts the Cilk-M [238] runtime scheduler to support on-the-fly pipeline parallelism using a bind-to-element approach. This chapter makes the following contributions:

- We describe linguistics for Cilk-P that allow on-the-fly pipeline parallelism to be incorporated into the Cilk fork-join parallel programming model (Section 7.2).
- We illustrate how Cilk-P linguistics can be used to express the *x264* benchmark as a pipeline program (Section 7.3).
- We characterize the execution dag of a Cilk-P pipeline program as an extension of a fork-join program (Section 7.4).
- We introduce the PIPER scheduling algorithm, a theoretically sound randomized work-stealing scheduler (Section 7.5).
- We prove that PIPER is asymptotically efficient, executing Cilk-P programs on P processors in $T_P \leq T_1/P + O(T_\infty)$ expected time (Sections 7.6 and 7.7).
- We bound space usage, proving that PIPER on P processors uses $S_P \leq P(S_1 + fDK)$ stack space for pipeline iterations, where S_1 is the serial stack space, f is the "frame size" (roughly, the maximum number of bytes consumed on the stack by any one pipeline iteration), D is the depth of nested pipelines, and K is the throttling limit (Section 7.8).
- We describe our implementation of PIPER in the Cilk-P runtime system, introducing two key optimizations, called "lazy enabling" and "dynamic dependency folding," for reducing synchronization overhead when two consecutive pipeline iterations execute in parallel (Section 7.9).
- We demonstrate that the *ferret*, *dedup*, and *x264* benchmarks from PARSEC that have been hand-compiled for the Cilk-P runtime system run competitively with existing Pthreaded implementations (Section 7.10).
- We prove two theorems regarding the performance impact of throttling (Section 7.11).

First we show that, if each stage has approximately the same cost and dependencies in every iteration, then throttling only reduces the parallelism in a pipeline by a constant factor. We then show that, if the cost of a stage can vary dramatically between iterations, however, then it is impossible for any scheduler to achieve parallel speedup when executing the pipeline without using a large amount of space.

We conclude in Section 7.12 with a discussion of potential future work.

7.2 On-the-fly pipeline programs

Cilk-P's linguistic model supports both fork-join and pipeline parallelism, which can be nested arbitrarily. For convenience, we shall refer to programs containing nested fork-join and pipeline parallelism simply as ***pipeline programs***. Cilk-P's on-the-fly pipelining model allows the programmer to specify a pipeline whose structure is determined during the pipeline's execution. This section shows how on-the-fly parallelism is supported in Cilk-P using a "`pipe_while`" construct.

To support on-the-fly pipeline parallelism, Cilk-P provides a `pipe_while` keyword. A `pipe_while` loop is similar to a serial `while` loop, except that loop iterations can execute in parallel in a pipelined fashion. The body of the `pipe_while` can be subdivided into stages, with stages named by user-specified integer values that strictly increase as the iteration executes. Each stage can contain nested fork-join and pipeline parallelism.

The boundaries of stages are denoted in the body of a `pipe_while` using the special functions `pipe_stage` and `pipe_stage_wait`. These functions accept an integer ***stage argument***, which is the number of the next stage to execute and which must strictly increase during the execution of an iteration. Every iteration i begins executing stage 0, represented by node $(i, 0)$. While executing a node (i, j'), if control flow encounters a `pipe_stage(j)` or `pipe_stage_wait(j)` statement, where $j' < j$, then node (i, j') ends, and control flow proceeds to node (i, j). A `pipe_stage(j)` statement indicates that node (i, j) can start executing immediately, whereas a `pipe_stage_wait(j)` statement indicates that node (i, j) cannot start until node $(i - 1, j)$ completes. The `pipe_stage_wait(j)` in iteration i creates a cross edge from node $(i - 1, j)$ to node (i, j) in the pipeline dag. Thus, by design choice, Cilk-P imposes the restriction that cross-edge pipeline dependencies only go between adjacent iterations. As we shall see in Section 7.9, this design choice facilitates the "lazy enabling" and "dynamic dependency folding" runtime optimizations.

The `pipe_stage` and `pipe_stage_wait` functions can be used without an explicit stage argument. Omitting the stage argument while executing stage j corresponds to an implicit stage argument of $j + 1$, meaning that control moves onto the next stage.

Cilk-P's semantics for `pipe_stage` and `pipe_stage_wait` statements allow for ***stage skipping***, where execution in an iteration i can jump stages from node (i, j') to node (i, j), even if $j > j' + 1$. If control flow in iteration $i + 1$ enters node $(i + 1, j'')$ after a `pipe_stage_wait`, where $j' < j'' < j$, then we implicitly create a ***null node*** (i, j'') in the pipeline dag, which has no associated work and incurs no scheduling overhead. We implicitly insert stage edges from (i, j') to (i, j'') and from (i, j'') to (i, j), as well as a cross edge from (i, j'') to $(i + 1, j'')$ in the pipeline dag, to capture the dependencies on this null node.

7.3 On-the-fly pipelining of *x264*

To illustrate the use of Cilk-P's `pipe_while` loop, this section describes how to parallelize the *x264* video encoder [408].

We begin with a simplified description of *x264*. Given a stream $\langle f_0, f_1, \ldots \rangle$ of video frames to encode, *x264* partitions the frame into a two-dimensional array of "macroblocks" and encodes each macroblock. A macroblock in frame f_i is encoded as a function of the encodings of similar macroblocks within f_i and similar macroblocks in frames "near" f_i. A frame f_j is ***near*** a frame f_i if $i - b \leq j \leq i + b$ for some constant b. In addition, we define a macroblock (x', y') to be ***near*** a macroblock (x, y) if $x - w \leq x' \leq x + w$ and $y - w \leq y' \leq y + w$ for some constant w.

The type of a frame f_i determines how a macroblock (x, y) in f_i is encoded. If f_i is an ***I-frame***, then macroblock (x, y) can be encoded using only ***previous*** macroblocks within f_i — macroblocks at positions (x', y') where $y' < y$ or $y' = y$ and $x' < x$. If f_i is a ***P-frame***, then macroblock (x, y)'s encoding can also be based on nearby macroblocks in nearby preceding frames, up to the most recent preceding I-frame,[2] if one exists within the nearby range. If f_i is a ***B-frame***, then macroblock (x, y)'s encoding can also be based on nearby macroblocks in nearby preceding or succeeding frames, specifically, frames in the interval between the most recently preceding I-frame and the next succeeding I- or P-frame.

Based on these frame types, an *x264* encoder must ensure that frames are processed in a valid order such that dependencies between encoded macroblocks are satisfied. A parallel *x264* encoder can pipeline the encoding of I- and P-frames in the input stream, processing each set of intervening B-frames after encoding the latest I- or P-frame on which the B-frame may depend.

Figure 7-2 shows Cilk-P pseudocode for an *x264* linear pipeline. Conceptually, the *x264* pipeline begins with a serial stage (lines 8–17) that reads frames from the input stream and determines the type of each frame. This stage buffers all B-frames at the head of the input stream until it encounters an I- or P-frame. After this initial stage, s hybrid stages process this I- or P-frame row by row (lines 18–25), where s is the number of rows in the video frame. After all rows of this I- or P-frame have been processed, the `PROCESS_BFRAMES` stage processes all B-frames in parallel (lines 27–29), and then the `END` stage updates the output stream with the processed frames (line 31).

Two issues arise with this general pipelining strategy, both of which can be handled using on-the-fly pipeline parallelism. First, the encoding of a P-frame must wait for the encoding of rows in the previous frame to be completed, whereas the encoding of an I-frame need not. These conditional dependencies are implemented in lines 20–24 of Figure 7-2 by executing a `pipe_stage_wait` or `pipe_stage` statement conditionally based on the frame's type. In contrast, many construct-and-run pipeline mechanisms assume that the dependencies on a stage are fixed for the entirety of a pipeline's execution, making such dynamic dependencies more difficult to handle. Second, the encoding of a macroblock in row x of P-frame f_i may depend on the encoding of a macroblock in a later row $x + w$ in the preceding I- or P-frame f_{i-1}. The code in Figure 7-2 handles such offset dependencies on line 17 by skipping w additional stages relative to the previous iteration. A similar stage-skipping trick is used on line 26 to ensure that the processing of a P-frame in iteration i depends only on the processing of the previous I- or P-frame, and not on the processing of preceding B-frames. Figure 7-3 illustrates the pipeline dag corresponding to the execution of the code in Figure 7-

[2]To be precise, up to a particular type of I-frame called an ***IDR-frame***.

```
01 // Symbolic names for important stages
02 const uint64_t PROCESS_IPFRAME = 1;
03 const uint64_t PROCESS_BFRAMES = 1 << 40;
04 const uint64_t END = PROCESS_BFRAMES + 1;
05 int i = 0;
06 int w = mv_range/pixel_per_row;
07
08 pipe_while(frame_t *f = next_frame()) {
09   vector<frame_t *> bframes;
10   f->type = decide_frame_type(f);
11   while(f->type == TYPE_B) {
12     bframes.push_back(f);
13     f = next_frame();
14     f->type = decide_frame_type(f);
15   }
16   int skip = w * i++;
17   pipe_stage_wait(PROCESS_IPFRAME + skip);
18   while(mb_t *macroblocks = next_row(frame)) {
19     process_row(macroblocks);
20     if(f->type == TYPE_I) {
21       pipe_stage;
22     } else {
23       pipe_stage_wait;
24     }
25   }
26   pipe_stage(PROCESS_BFRAMES);
27   cilk_for(int j = 0; j < bframes.size(); ++j) {
28     process_bframe(bframes[j]);
29   }
30   pipe_stage_wait(END);
31   write_out_frames(frame, bframes);
32 }
```

Figure 7-2: Example C++-like pseudocode for the *x264* linear pipeline. This pseudocode uses Cilk-P's linguistics to define hybrid pipeline stages on the fly, specifically with the `pipe_stage_wait` on line 17, the input-data dependent `pipe_stage_wait` or `pipe_stage` on lines 20–24, and the `pipe_stage` on line 26.

2, assuming that $w = 1$. Skipping stages shifts the nodes of an iteration down, adding null nodes to the pipeline, which do not increase the work or span.

7.4 Computation-dag model

Although the pipeline-dag model provides intuition for programmers to understand the execution of a pipeline program, it is not precise enough to prove theoretical performance guarantees. For example, a pipeline dag has no real way of representing nested fork-join or pipeline parallelism within a node. This section describes how to represent the execution of a pipeline program as a more refined computation dag. First, we present an example of a simple pipeline program using `pipe_while` loops, and explain how to transform it into an ordinary Cilk program with special function calls to enforce non-fork-join dependencies. Then we describe how to model these transformed programs as computation dags.

Intuitively, we shall model an execution of a pipeline program as a ***(pipeline) computation dag*** by augmenting a traditional dag model for fork-join parallel computations (described in Chapter 2) with cross and throttling dependencies. More formally, to generate a pipeline computation dag for an arbitrary pipeline-program execution, we use the following three-step process:

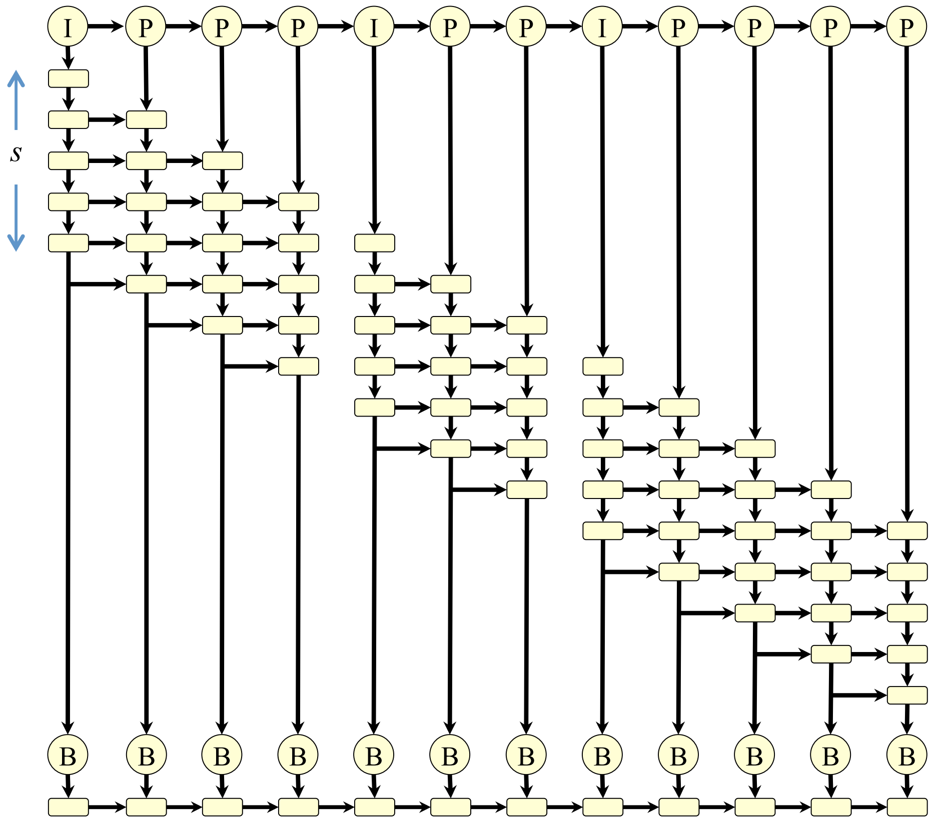


Figure 7-3: The pipeline dag generated for *x264*. Each iteration processes either an I- or P-frame, each consisting of s rows. As the iteration index i increases, the number of initial stages skipped in the iteration also increases. This stage skipping produces cross edges into an iteration i from null nodes in iteration $i-1$. Null nodes are represented as the intersection between two edges.

1. Transform the executed code in each `pipe_while` loop into ordinary Cilk code, augmented with special functions to implement cross and throttling dependencies.
2. Model the execution of this augmented Cilk program as a fork-join computation dag, ignoring cross and throttling dependencies.
3. Augment the fork-join computation dag with cross and throttling edges derived from the special functions.

The remainder of this section examines each of these steps in detail.

Code transformation for a `pipe_while` *loop*

Let us first consider the process of translating a `pipe_while` loop into ordinary Cilk code. Conceptually, a `pipe_while` loop is transformed into an augmented ordinary Cilk program in which an ordinary `while` loop sequentially spawns off each iteration of the `pipe_while` loop. In the body of this `while` loop, each iteration first executes stage 0. Upon executing the first `pipe_stage` or `pipe_stage_wait` instruction in iteration i, the remainder of iteration i is spawned off, allowing the remaining stages of iteration i to execute in parallel with iteration $i+1$. By executing stage 0 of a `pipe_while` iteration before spawning the remaining stages, stage 0 is ensured to execute sequentially across all iterations of the `while` loop. Each iteration can execute additional runtime functions to enforce cross and throttling dependencies between iterations.

This conceptual transformation of a `pipe_while` loop is complicated by specific semantic

```
33 int fd_out = open_output_file();
34 bool done = false;
35 pipe_while (!done) {
36   chunk_t *chunk = get_next_chunk();
37   if (chunk == NULL) {
38     done = true;
39   } else {
40     pipe_stage_wait(1);
41     bool isDuplicate = deduplicate(chunk);
42     pipe_stage(2);
43     if (!isDuplicate)
44       compress(chunk);
45     pipe_stage_wait(3);
46     write_to_file(fd_out, chunk, isDuplicate);
47   }
48 }
```

Figure 7-4: Cilk-P pseudocode for the parallelization of the *dedup* compression program as an SSPS pipeline.

features of `pipe_while` iterations. For example, an ordinary `cilk_spawn` creates a ***(function) frame*** — activation record — for the spawned child, and then immediately spawns the child subcomputation. For a `pipe_while` iteration, however, although stage 0 of each iteration executes before the remaining stages of each iteration are spawned, the runtime executes each iteration, including stage 0, within its own frame to ensure that all stages of an iteration operate on the same set of iteration-local variables. Furthermore, to ensure that an iteration executes pipeline stages sequentially, the runtime executes an implicit `cilk_sync` at the end of each stage, which syncs all child functions spawned within the stage before allowing the next stage to begin.

To more precisely illustrate the semantic features of `pipe_while` iterations, including how the Cilk-P runtime manages frames and iterations of a `pipe_while` loop, let us consider a Cilk-P implementation of a specific pipeline program, namely, the *dedup* compression program from PARSEC [45, 46]. The benchmark can be parallelized by using a `pipe_while` to implement an SSPS pipeline. Figure 7-4 shows Cilk-P pseudocode for *dedup*, which compresses the provided input file by removing duplicated "chunks," as follows. Stage 0 (lines 36–38) of the program reads data from the input file and breaks the data into chunks (line 36). As part of stage 0, it also checks the loop-termination condition and sets the `done` flag to `true` (line 38) if the end of the input file is reached. If there is more input to be processed, the program begins stage 1 (line 41), which calculates the SHA1 signature of a given chunk and queries a hash table whether this chunk has been seen using the SHA1 signature as key. Stage 1 is a serial stage as dictated by the `pipe_stage_wait` on line 40. Stage 2 (line 44), which the `pipe_stage` on line 42 indicates is a parallel stage, compresses the chunk if it has not been seen before. The final stage (line 46) is a serial stage that writes either the compressed chunk or its SHA1 signature to the output file depending on whether it is the first time the chunk has been seen.

Figure 7-5 illustrates how the Cilk-P runtime system manages frames and pipeline iterations for the `pipe_while` loop for *dedup* presented in Figure 7-4. This code transformation has six key components, which illustrate the general structure of parallelism in pipeline programs.

1 As shown in lines 51–95, a `pipe_while` loop is "lifted" using a C++ lambda function [376, Sec.11.4] and converted to an ordinary `while` loop whose iterations correspond to iterations of the pipeline. This lambda function declares a ***control frame*** object `pcf`

```
49 int fd_out = open_output_file();
50 bool done = false;
51 [&]() {
52   _Cilk_pipe_control_frame pcf(0);
53   while (true) {
54     _Cilk_pipe_iter_frame* next_iter_f = pcf.get_new_iter_frame(pcf.i);
55     // Stage 0 of an iteration.
56     [&]() {
57         next_iter_f->continue_after_stage0 = false;
58         if (!done) {
59           next_iter_f->chunk = get_next_chunk();
60           if (next_iter_f->chunk == NULL)
61             done = true;
62           else
63             next_iter_f->continue_after_stage0 = true;
64         }
65         cilk_sync;
66     }();
67     // Spawn the remaining stages of iteration pcf.i, if they exist.
68     if (next_iter_f->continue_after_stage0) {
69       cilk_spawn [&](_Cilk_pipe_iter_frame* iter_f) {
70         // assert(iter_f->stage_counter < 1);
71         iter_f->stage_counter = 1;
72         // node (i,1) begins
73         iter_f->stage_wait(1);
74         iter_f->isDuplicate = deduplicate(iter_f->chunk);
75         cilk_sync;
76         // assert(iter_f->stage_counter < 2);
77         iter_f->stage_counter = 2;
78         // node (i,2) begins
79         if (!iter_f->isDuplicate) compress(iter_f->chunk);
80         cilk_sync;
81         // assert(iter_f->stage_counter < 3);
82         iter_f->stage_counter = 3;
83         // node (i,3) begins
84         iter_f->stage_wait(3);
85         write_to_file(fd_out, iter_f->chunk, iter_f->isDuplicate);
86         cilk_sync;
87         iter_f->stage_counter = INT64_MAX;
88       }(next_iter_f);
89     } else break;
90     // Advance to next iteration and check for throttling.
91     pcf.i++;
92     pcf.throttle(pcf.i - pcf.K);
93   }
94   cilk_sync;
95 }();
```

Figure 7-5: Pseudocode resulting from translating the execution of the Cilk-P *dedup* implementation from Figure 7-4 into Cilk Plus code augmented by cross and throttling dependencies, implemented by `iter_f->stage_wait` and `pcf.throttle`, respectively. The unbound variable `pcf.K` is the throttling limit.

(on line 52) to keep track of runtime state needed for the `pipe_while` loop, including a variable `pcf.i` to index iterations, which line 52 initializes to 0.

2 Each iteration of the `while` loop allocates an ***iteration frame*** to store local data for each pipeline iteration. Before starting a pipeline iteration `pcf.i`, the loop allocates a new iteration frame `next_iter_f` for iteration `pcf.i`, as shown in line 54. The iteration frame stores local variables declared in the body of an iteration that persist across pipeline stages. For *dedup*, for example, Figure 7-4 shows that the local variable `chunk`

is used through all stages. The iteration frame also stores a ***stage-counter*** variable, `iter_f->stage_counter`, to track the currently executing stage for the iteration. Although Figure 7-5 shows iteration frames of a generic type `Cilk_pipe_iter_frame`, in practice, a compiler would generate a unique iteration frame type for each `pipe_while` loop body, since the local variables stored in the frame are specific to the loop body.

3 The body of this `while` loop is split into two nested lambda functions, the first for stage 0 of the iteration (lines 56–66), and the second for the remaining stages in the iteration (lines 69–88), if they exist. This transformation guarantees that stage 0 is always a serial stage, since the first lambda function is directly called in the body of the `while` loop. The test condition of the `pipe_while` loop is evaluated as part of stage 0, as demonstrated in line 58. In contrast, the `cilk_spawn` in line 69 allows the remaining stages of an iteration to execute in parallel with the next iteration of the loop. The `cilk_sync` immediately after the end of the `while` loop (line 94) ensures that all spawned iterations complete before the `pipe_while` loop finishes.

4 The last statement in the `while` loop (line 92) is a call to a special function `throttle`, defined by the control frame `pcf`, which enforces the throttling dependency that iteration `pcf.i` can not start until iteration `pcf.i - pcf.K` has completed.

5 A `pipe_stage` statement in the original `pipe_while` loop is transformed into an update to `iter_f->stage_counter`, while a `pipe_stage_wait` statement is transformed into an update followed by a call to `iter_f->stage_wait`, which enforces the cross dependency on the previous iteration. In *dedup*, stages 1, 2, and 3 are thus delineated by updates to `iter_f->stage_counter` in lines 71, 77, and 82, respectively. The end of the iteration is delineated by setting `iter_f->stage_counter` to its maximum value, such as in line 87.

6 At the end of each stage (lines 65, 75, 80, and 86), a `cilk_sync` guarantees that any nested fork-join parallelism is enclosed within the stage, that is, any functions spawned in `cilk_spawn` statements within the stage, return before the next stage begins.

For *dedup*, Figure 7-5 is able to use lambda functions to capture the parallel control structure of Figure 7-4 directly in Cilk, without changing the semantics of the `cilk_spawn` or `cilk_sync` keywords. This transformation introduces an additional variable in the iteration frame, `continue_after_stage0`, so that execution can resume correctly at the continuation of stage 0 in the second lambda function in each iteration. The lambda functions in line 51 and line 56 exist only to create nested scopes for parallelism and ensure the desired behavior for a `cilk_sync` statement. Without the lambda function in line 51, the last `cilk_sync` in line 94 would also synchronize with any functions that were spawned in the enclosing function before calling the `pipe_while` loop. Similarly, the lambda for stage 0 in line 56 exists only to guarantee that the `cilk_sync` in line 65 joins only the parallelism within stage 0, and not with any of the lambda functions spawned in line 69. All the lambda functions in Figure 7-5 capture the environment of the enclosing function by reference because the body of the `pipe_while` loop is allowed to access variables declared in the enclosing function, such as `fd_out` and `done`.

While Figure 7-5 illustrates the semantics that Cilk-P requires for compiling the code in Figure 7-4, it is not intended to be a complete compiler transformation for `pipe_while` loops. In practice, we expect it to be simpler and more efficient for compiler transforms for `pipe_while` loops to operate and produce output at a lower level than ordinary Cilk code. For example, a more complicated pipeline iteration in which stage 0 may end in the middle of a loop is tricky to express using lambdas and ordinary Cilk code, because of the scoping of local variables and the semantics of `cilk_sync` statements. At a lower level,

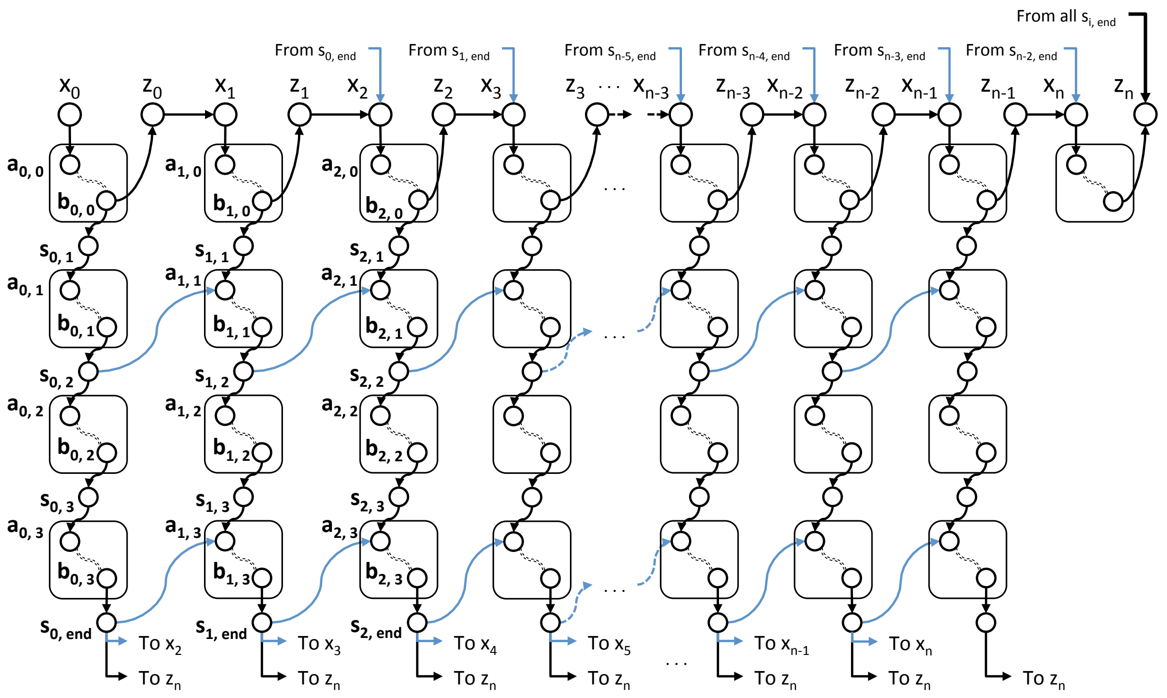


Figure 7-6: An example pipeline computation dag for a pipe_while loop with n iterations, $m = 4$ stages, and throttling limit $K = 2$, corresponding to the transformation shown in Figure 7-5. The vertices are organized to reflect their organization in a pipeline dag. Each rounded box contains the vertices corresponding to the execution of a node. A double-dashed line indicates a computation subdag whose structure is not shown. A column of these boxes corresponds to an iteration of the pipe_while, while a row of these boxes corresponds to a stage. Additional vertices and edges appear in this dag to denote instructions executed by the runtime to handle iterations of a pipe_while, as well as their parallel control dependencies. Cross and throttling edges are colored blue, while edges in typical Cilk programs are colored black.

however, a compiler could handle the end of stage 0 by directly generating code that saves the program state analogously to an ordinary cilk_spawn. The compiler might also be able to eliminate one or more of the nested lambda functions from Figure 7-5, and instead generate code directly for modified versions of cilk_sync and cilk_spawn statements specifically for pipe_while loop transformations. The generated code would not be directly expressible in Cilk, but would likely be simpler and more efficient.

Similarly, although Figure 7-5 describes an iteration as being split into two lambda functions — one for stage 0 and one for the subsequent stages of the iteration — in practice, it may be simpler to merge those lambda functions. Instead of using an ordinary cilk_spawn to spawn the rest of the stages of an iteration separately from stage 0, for example, a system might instead try to spawn a single lambda function for the entire iteration. Then, the system might allow other workers to steal the continuation of the spawn of the iteration only after the iteration finishes its stage 0, not immediately after the spawn occurs.

Pipeline computation dag for dedup

Given the transformed code for a pipe_while loop, the second and third steps generate a pipeline computation dag that models the execution of this transformed loop. The second step models the execution of the transformed code when ignoring all calls to stage_wait and throttle, and then the third step augments the resulting fork-join computation dag with cross and throttling edges derived from those calls. Figure 7-6 illustrates the salient features of the final pipeline computation dag that corresponds to executing the code in Figure 7-5. Let us examine the structure of the dag in Figure 7-6 by first considering the vertices and edges that model the execution of Figure 7-5, ignoring calls to `stage_wait` and `throttle`, and then examining the cross and throttling edges added by these calls.

Let us first see how the vertices in Figure 7-6 correspond to the lines of code in Figure 7-5. Let i and j be integers where $0 \leq i \leq n$ and j is nonnegative.

The vertices labeled x_i and z_i correspond to the execution of instructions inserted by the runtime. Vertices x_0 and z_n correspond to the first and final instruction, respectively, of the lambda for the `pipe_while` loop. Vertex x_0, which is called the ***pipeline root***, corresponds to executing line 52 to create the control frame for the `pipe_while` loop, while vertex z_n, which is called the ***pipeline terminal***, corresponds to executing line 94. The remaining vertices z_i and x_i are called ***iteration-increment*** and ***iteration-throttle*** vertices, respectively. Between each iteration of the `while` loop, the iteration-increment vertices z_i correspond to executing line 91, and the iteration-throttle vertices x_i correspond to executing line 92.

The computation subdag rooted at $a_{i,j}$ and terminated at vertex $b_{i,j}$ corresponds to the execution of node (i, j) in the pipeline dag. These computation subdags are guaranteed to each have a single root and a single terminal, through the use of `cilk_sync` statements at the end of each stage. We call vertex $a_{i,j}$ the ***node root***, and we call vertex $b_{i,j}$ the ***node terminal***.

The computation subdag for node $(i, 0)$ corresponds to executing stage 0 and associated runtime instructions for managing the `while` loop in iteration i. Node root $a_{i,0}$ corresponds to executing line 53. Node terminal $b_{i,0}$ corresponds to executing the `cilk_spawn` statement on line 69, except when $i = n$, in which case $b_{n,0}$ corresponds to executing line 89. The vertices in Figure 7-6 on paths from $a_{i,0}$ to $b_{i,0}$ correspond to executing the intervening instructions in lines 53–69. The `cilk_sync` statement in the lambda for stage 0 ensures that vertex $b_{i,0}$ is the single leaf vertex for this computation subdag.

The computation subdag for node (i, j) for $i < n$ and $j > 0$ is similar to that for node $(i, 0)$. For example, in iteration i, node root $a_{i,1}$ corresponds to executing line 73 — the first instruction in node $(i, 1)$ — and node terminal $b_{i,1}$ corresponds to executing line 75 — the final instruction in node $(i, 1)$. The vertices on paths from $a_{i,1}$ to $b_{i,1}$, in Figure 7-6, correspond to executing the intervening instructions in lines 73–75. Notice that, if node (i, j) is the destination of a cross edge, then $a_{i,j}$ corresponds to executing `stage_wait`. The `cilk_sync` statement at the end of each stage — lines 75, 80, and 86 for stages 1, 2, and 3, respectively — ensure that $b_{i,j}$ is the single leaf in the computation subdag corresponding to the execution of node (i, j).

The ***stage-counter*** vertices $s_{i,end}$ and $s_{i,j}$ for $j > 0$ correspond to updates in iteration i to the iteration frame's `stage_counter` variable. For example, $s_{i,2}$ corresponds to executing line 77 in iteration i. Vertex $s_{i,end}$ corresponds to executing line 87 in iteration i, which terminates the iteration. We call $s_{i,end}$ the ***terminal*** vertex for iteration i.

The correspondence between instructions in Figure 7-5 and the vertices of Figure 7-6 describes most of the edges in Figure 7-6, based on the structure of fork-join computation dags. For example, the code in Figure 7-5 shows that, for each i where $0 \leq i \leq n$, edge $(x_i, a_{i,0})$ is a serial edge, edge $(b_{i,0}, s_{i,1})$ is a spawn edge, and edge $(b_{i,0}, z_i)$ is a continue edge. Meanwhile, for each iteration i where $0 \leq i < n$, edge (z_i, x_{i+1}) is a serial edge, reflecting the fact that stage 0 is a serial stage. Similarly, for $j > 1$, the edges $(b_{i,j-1}, s_{i,j})$ and $(s_{i,j}, a_{i,j})$ that connect the node terminal of $(i, j - 1)$ to the node root of (i, j) are serial edges, reflecting the fact that each iteration of the pipeline executes the pipeline stages sequentially. Finally, for each iteration i where $0 \leq i < n$, edge $(b_{i,3}, s_{i,end})$ is a serial edge, and edge $(s_{i,end}, z_n)$ is a return edge. These vertex and edge definitions are established by modeling an execution of the transformed code as an ordinary Cilk program, when `stage_wait` and `throttle` instructions are ignored.

Finally, we consider the cross and throttling edges in Figure 7-6 enforced by `stage_wait` and `throttle` instructions.

For each iteration i where $0 < i < n$, a call to `stage_wait` implements a cross edge, which connects a stage-counter vertex in iteration $i-1$ to a node root in iteration i. For example, in each iteration i of the loop in Figure 7-5, the `stage_wait` call on line 73 implements the cross edge $(s_{i-1,2}, a_{i,1})$, and the `stage_wait` call on line 84 implements the cross edge $(s_{i-1,end}, a_{i,3})$. Conceptually, because a stage-counter vertex $s_{i,j}$ occurs after the node terminal for stage $j-1$ and before the node root for stage j, a cross edge $(s_{i-1,j}, a_{i,j-1})$ ensures that node $(i, j-1)$ in iteration i executes after node $(i-1, j-1)$. When j is the final stage in an iteration $i-1$, the iteration terminal $s_{i-1,end}$ fills the role of the stage-counter vertex $s_{i-1,j+1}$.

A throttling edge connects the terminal of iteration $i \leq n-K$ to the iteration-throttle vertex x_{i+K} in iteration $i+K$, where K is the throttling limit. Figure 7-6 illustrates throttling edges when $K = 2$ and shows that a throttling edge exists from $s_{i,end}$ to x_{i+2} for each iteration i where $0 \leq i < n-2$. These throttling edges thus prevent node $(i, 0)$ from executing before all nodes in iteration $i-K$ complete, thereby limiting the number of iterations that may execute simultaneously.

General pipeline computation dags

To generalize the structure of the pipeline computation dag in Figure 7-6 for arbitrary Cilk-P pipelines, we must specify how null nodes are handled. In some iteration i, for stage $j > 0$, suppose that node (i, j) is a null node. In this case, none of the stage-counter vertices $s_{i,j}$, node roots $a_{i,j}$, node terminals $b_{i,j}$, nor any of the vertices on paths between these, map to executed instructions, and therefore these vertices do not exist in the computation dag. To demonstrate what happens to the edges that would normally enter and exit these vertices, we can pretend that the computation dag is originally constructed with dummy vertices $s_{i,j}$, $a_{i,j}$, and $b_{i,j}$ connected in a path, and then all three of these vertices are contracted into the stage-counter vertex following $b_{i,j}$. Because $a_{i,j}$ is a dummy vertex, it does not correspond to a call to `stage_wait`, and thus it has no incoming cross edge. Furthermore, this method for handling null nodes can cause multiple cross edges to exit the same stage-counter vertex. We shall see that this is does not pose a problem for the PIPER scheduler.

7.5 The PIPER scheduler

PIPER executes a pipeline program on a set of P workers using work-stealing, as this section describes in detail. For the most part, PIPER's execution model can be viewed as a modification of the scheduler described by Arora, Blumofe, and Plaxton [24] (henceforth referred to as the ABP model) for computation dags arising from pipeline programs. PIPER deviates from the ABP model in one significant way, however, in that it performs a "tail-swap" operation, a special operation introduced to handle throttling of pipeline iterations.

We describe the operation of PIPER in terms of the pipeline computation dag $G = (V, E)$. Each worker p in PIPER maintains an ***assigned vertex*** corresponding to the instruction that p executes on the current time step. We say that a vertex u is ***ready*** if all its predecessors have been executed. Executing an assigned vertex v can ***enable*** a vertex u that is a direct successor of v in G by making u ready. Each worker maintains a deque of ready vertices. Normally, a worker pushes and pops vertices from the tail of its deque. A thief, however, may try to steal a vertex from the head of another worker's deque. It is convenient to define the ***extended deque*** $\langle v_0, v_1, \ldots, v_r \rangle$ of a worker p, where $v_0 \in V$ is p's assigned vertex and $v_1, v_2, \ldots, v_r \in V$ are the vertices in p's deque in order from tail to head.

On each time step, each Piper worker p follows a few simple rules for execution based on the type of p's assigned vertex v and how many direct successors are enabled by the execution of v, which is at most 2. In particular, although v may have multiple immediate successors in the next iteration due to cross-edge dependencies from null nodes, executing v can enable at most one such vertex, because the nodes in the next iteration execute serially. To simplify the mathematical analysis, we assume that all rules are executed atomically.[3]

We first consider the cases where the assigned vertex v of a worker p is not the terminal of an iteration.

- If executing v enables only one direct successor u, then p simply changes its assigned vertex from v to u.
- If executing v enables two successors u and w, then p changes its assigned vertex from v to one successor u, and pushes the other successor w onto its deque. The decision of which successor to push onto the deque depends on the type of v. If v is a vertex corresponding to a normal spawn, then u follows the spawn edge (u is the child), and w follows the continue edge (w is the continuation).[4] If v is a stage-counter vertex in iteration i that is not the terminal of iteration i, then u is the node root of the next node in iteration i, and w is the node root of a node in iteration $i + 1$.
- If executing v enables no successors and the deque of p is not empty, then p pops the bottom element u from its deque and changes its assigned vertex from v to u.
- If executing v enables no successors and the deque of p is empty, then p becomes a thief. As a thief, p randomly picks another worker to be its victim and tries to steal the vertex at the head of the victim's deque. If such a vertex u exists, the thief p sets its assigned vertex to u. Otherwise, the victim's deque is empty, p's assigned node becomes null, and p remains a thief.

These cases are consistent with the normal ABP model.

Piper handles the terminal of an iteration a little differently, because of throttling edges. Suppose that a worker p has an assigned vertex v which is the terminal of an iteration. When p executes v, it can enable some combination of a pipeline terminal, an iteration-throttle vertex, or the destination a cross edge, if there is a cross edge leaving v. We can distinguish these cases, however, based on whether or not v enables an iteration-throttle vertex.

- Suppose that executing v does not enable an iteration-throttle vertex. Then, p acts as it would in the normal ABP model. In particular, p can only enable either the destination a of a cross edge or a pipeline terminal z, if it enables any vertex. In this situation, p executes as it normally would when enabling zero or one vertices.
- Suppose that executing v does enable an iteration-throttle vertex x. If executing v

[3]This assumption, which is also implicit in the ABP model, need not correspond to an actual implementation. Work-stealing schedulers can typically optimize away many atomic operations by using more sophisticated synchronization.

[4]This choice to have a worker follow the spawn edge instead a continue edge is consistent with traditional "parent-stealing" execution model of Cilk, which on a spawn begins executing the spawned child function and leaves the continuation in the parent frame to be stolen. Unlike child stealing, parent stealing allows us to prove a bound on space in Section 7.8.

also enables the destination a of a cross edge, then p behaves as it would in the normal ABP model, pushing x onto its deque and setting its assigned vertex to a. If executing v does not enable the destination of a cross edge, then p performs two actions. First, p changes its assigned vertex from v to x. Second, if p has a nonempty deque, then p performs a ***tail swap***: it exchanges its assigned vertex x with the vertex at the tail of its deque.

This tail-swap operation is designed to empirically reduce PIPER's space usage and cause PIPER to favor retiring old iterations over starting new ones. Without the tail swap, in a normal ABP-style execution, when a worker p finishes an iteration i that enables a vertex via a throttling edge, p would conceptually choose to start a new iteration $i + K$, even if iteration $i + 1$ were already suspended and on its deque. With the tail swap, p resumes iteration $i+1$, leaving $i+K$ available for stealing. The tail swap also enhances cache locality by encouraging p to execute consecutive iterations.

It may seem, at first glance, that a tail-swap operation might significantly reduce the parallelism, since the vertex z enabled by the throttling edge is pushed onto the bottom of the deque. Intuitively, if there were additional work above z in the deque, then a tail swap could significantly delay the start of iteration $i + K$. Lemma 44 in Section 7.6 will show, however, that a tail-swap operation only occurs on deques with exactly 1 element. Thus, whenever a tail swap occurs, z is at the top of the deque and is immediately available to be stolen.

7.6 Structural invariants

During the execution of a pipeline program by PIPER, the worker deques satisfy two structural invariants, called the "contour" property and the "depth" property. This section states and proves these invariants.

Intuitively, we would like to describe the structure of the worker deques in terms of frames, because these frames implement a cactus stack [178, 238] that reflects the parallel control structure of the program. In variants of Cilk, every spawned function creates a new frame, and every iteration of a parallel loop executes in its own frame. The cactus stack ensures that every function is allowed to access the variables in its frame and its parent frames, in spite of their parallel execution. For non-pipelined computation dags, the creation of new frames matches the parallel control structure, and pushing a vertex onto a worker's deque corresponds to creating a new frame.

We would like to have an analogous structure for `pipe_while` loops. As Section 7.4 describes, PIPER creates a new frame for each iteration and executes all stages of the iteration using that frame. The creation of these per-iteration frames does not match how Piper manipulates the worker deques, however. In particular, stage 0 of an iteration executes using that iteration's frame before the remaining stages of the iteration are spawned. Consequently, the iteration's frame is created before stage 0 executes, but no vertex from this iteration is pushed onto a worker's deque until after stage 0 completes.

To get around this problem with frames, we introduce "contours" to model how deques are modified during the execution of a `pipe_while` loop. Consider a computation dag $G = (V, E)$ that arises from executing a pipeline program. A ***contour*** is a path in G composed only of serial and continue edges. A contour must be a path, because there can be at most one serial or continue edge entering or leaving any vertex. We call the first vertex of a contour the ***root*** of the contour, which is the only vertex in the contour that has an incoming spawn edge (except for the initial instruction of the entire computation, which has no incoming edges). Consequently, contours can be organized into a tree hierarchy, where one contour is a parent of another if the first contour contains a vertex that spawns the root of the second. Given a vertex $v \in V$, let $c(v)$ denote the contour to which v belongs. For convenience, we shall assume that all contours are maximal, meaning that no two vertices in distinct contours are connected by a serial or continue edge.

Figure 7-7 illustrates contours for a simple function `F` with both nested fork-join and pipeline parallelism. For the `pipe_while` loop in `G`, stage 0 of pipeline iteration 0 (a_7 and a_8) is considered part of the same contour that starts the `pipe_while` loop, not part of contour f which represents the rest of the stages of iteration 0. In terms of function frames, however, it is natural to consider stage 0 as sharing a function frame with the rest of the stages in the same iteration, as Section 7.4 describes. Although contour boundaries happen to align with function boundaries when we consider only fork-join parallelism in Cilk, `pipe_while` loops highlight the fact that contours and function frames are actually distinct, orthogonal concepts.

One important property of contours, which can be shown by structural induction, is that for any function invocation `F` the vertices p and q corresponding to the first and last instructions in `F` belong to the same contour, that is, $c(p) = c(q)$. Using this property and the identities of its edges, one can show the following facts about contours in a pipeline computation dag.

FACT 1. For a given `pipe_while` loop on n iterations all of the following vertices lie in the same contour: the pipeline root x_0, pipeline terminal z_n, iteration-increment vertices $z_1, z_2, \ldots, z_{n-1}$, iteration-throttle vertices $x_1, x_2, \ldots, x_n$, the node roots $a_{0,0}, a_{1,0}, \ldots, a_{n,0}$ for stage 0, and the node terminals $b_{0,0}, b_{1,0}, \ldots, b_{n,0}$ for stage 0. In other words we have $c(x_i) = c(a_{i,0}) = c(b_{i,0}) = c(z_i)$ for all integers i where $0 \le i \le n$.

FACT 2. For an iteration i of a `pipe_while` loop and $j = 1, 2, \ldots, m-1$, if node (i,j) is not a null node, then the stage-counter vertices $s_{i,j}$ and $s_{i,end}$ and the node root $a_{i,j}$ and node terminal $b_{i,j}$ for node (i,j) all lie in the same contour, that is, $c(s_{i,end}) = c(s_{i,j}) = c(a_{i,j}) = c(b_{i,j})$.

For convenience, we say that the ***root*** of a `pipe_while` iteration i is the first stage-counter vertex in iteration i, specifically, the stage-counter vertex in iteration i that is the destination of a spawn edge from the stage-0 node terminal for iteration i. Consequently, the root of the contour containing the stage-counter vertices for an iteration i is the root of iteration i.

The following two lemmas describe two important properties exhibited in the execution of a pipeline program. The first lemma observes that no two distinct workers can contain vertices from the same contour at the same time in their deques.

Lemma 42 *Only one vertex in a contour can belong to any extended deque at any time.*

PROOF. The vertices in a contour form a chain and are, therefore, enabled serially. □

The second lemma describes how the endpoints of a cross edge or a throttling edge are related in terms of contours. In particular, the structure of a `pipe_while` guarantees that each iteration creates a separate contour for all its stages after stage 0, and that all contours for iterations of the `pipe_while` share a common parent in the contour tree, namely the contour containing stage 0 of all loop iterations.

```
void F(int n) {
  if (n < 2)
    G(n);
  else {
    cilk_spawn F(n-1);
    F(n-2);
    cilk_sync;
  }
}
```

```
void G(int n) {
  if (n == 0) {
    int i = 0;
    pipe_while(i < 2) {
      ++i; // Stage 0
      pipe_stage_wait(1);
      H(); // Stage 1.
    }
  }
}
```

```
void H() {
  cilk_spawn foo();
  bar();
  cilk_sync;
}
```

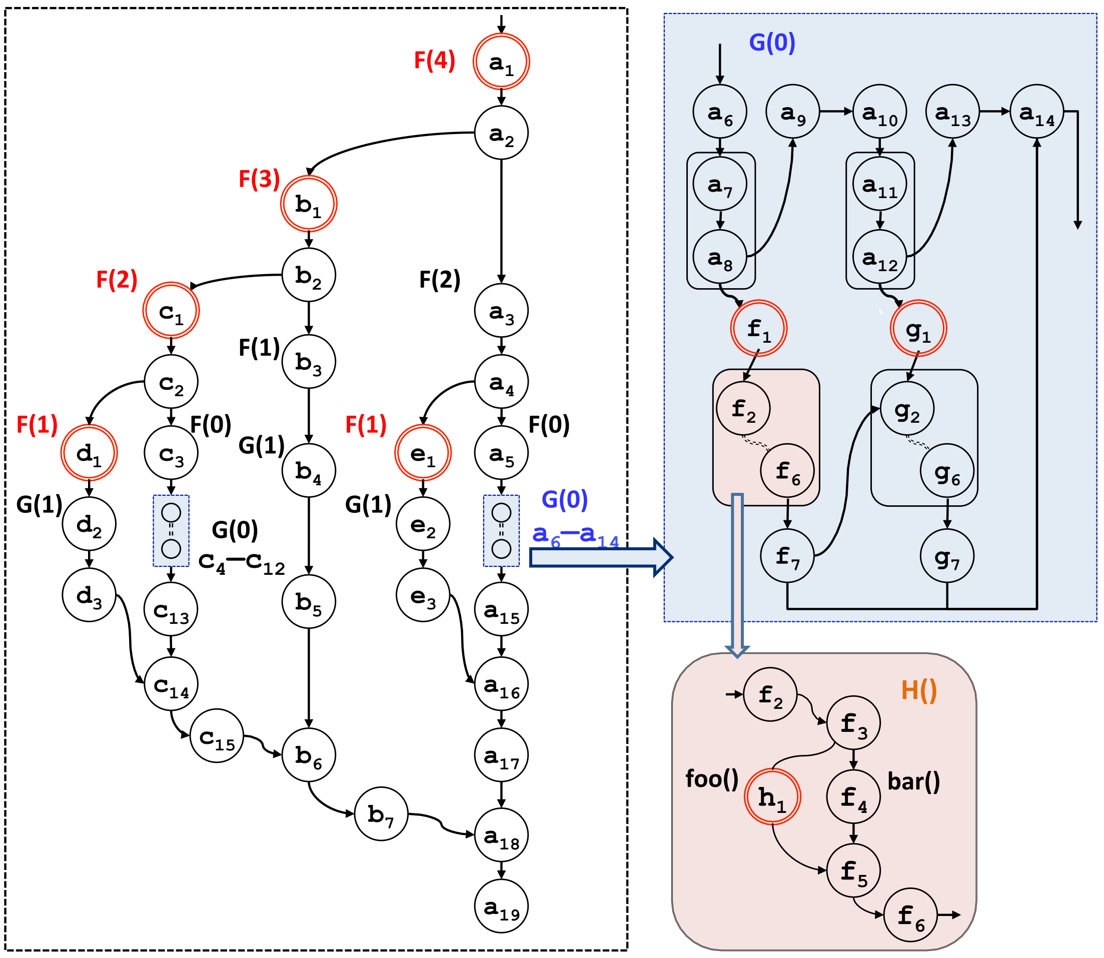


Figure 7-7: Contours for a computation with fork-join and pipeline parallelism. The fork-join function `F` contains nested calls to a function `G` that contains a `pipe_while` loop with two iterations. The function `G` itself calls a fork-join function `H` in stage 1 of each iteration. Each letter a through h labels a contour in the dag for `F(4)`. The vertices $a_1, b_1, \ldots, h_1$ are contour roots. For example, the root of $c(a_k)$ is a_1 for all k. Similar to Figure 7-6, the rounded rectangles in the subdag for `G(0)` represent pipeline stages. A double-dashed line represents an additional subdag whose structure is not shown.

Lemma 43 *If an edge (u, v) is a cross edge, then $c(u)$ and $c(v)$ are siblings in the contour tree and correspond to adjacent iterations in a* `pipe_while` *loop. If an edge (u, v) is a throttling edge, then $c(v)$ is the parent of $c(u)$ in the contour tree.*

Proof. This lemma follows naturally from the structure of pipeline computation dags. For $i > 0$, every cross edge (u, v) connects a stage-counter vertex in iteration $i-1$ (that is, u

equals either $s_{i-1,j}$ for some j or $s_{i-1,end}$) to a node root $v = a_{i,k}$ in iteration i in the same `pipe_while` loop. By Fact 2, the root of the contour $c(u)$ is the root of iteration $i-1$. Thus the contour $c(u)$ is a child of the contour $c(b_{i-1,0})$ in the contour tree. Similarly, the root of the contour $c(v) = c(a_{i,k})$ is a child of the contour $c(b_{i,0})$ containing the node terminal $b_{i,0}$. Because $b_{i-1,j}$ and $a_{i,k}$ belong to iterations of the same `pipe_while` loop, Fact 1 implies that $c(b_{i-1,0}) = c(b_{i,0})$. Because $c(u)$ and $c(v)$ are both children of this contour, $c(u)$ and $c(v)$ are siblings in the contour tree, showing the first part of the lemma.

For a `pipe_while` loop of n iterations with throttling limit K, a throttling edge (u, v) connects u, the terminal of an iteration $i < n-K$, to a vertex $v = x_{i+K}$ in the computation dag. By the reasoning above, u must be in a child contour of $c(b_{i,0})$. By Fact 1, we know $c(b_{i,0}) = c(x_{i+K}) = c(v)$. Thus, u is in a child contour of $c(v)$, showing the second part of the lemma. □

During Piper's execution of a pipeline program, the workers' deques are highly structured with respect to contours. As the following definition details, with one exception, the contours for two adjacent vertices in a worker's extended deque obey a strict parent-child relationship. The sole exception to this property is that, for any particular `pipe_while` loop, an extended deque can contain at most two vertices that belong to contours for iterations of that loop. These two vertices will be adjacent in the extended deque, and the contours that contain them will correspond to sibling `pipe_while` loop iterations.

Definition 3 *At any time during an execution of a pipeline program which produces a computation dag $G = (V, E)$, consider the extended deque $\langle v_0, v_1, \ldots, v_r\rangle$ of a worker p. This deque satisfies the **contour property** if, for all $k = 0, 1, \ldots r-1$, one of the following two conditions holds:*

1. *Contour $c(v_{k+1})$ is the parent of $c(v_k)$.*
2. *The root of $c(v_k)$ is the root for some iteration i, the root of $c(v_{k+1})$ is the root for the next iteration $i+1$, and if $k+2 \le r$, then $c(v_{k+2})$ is the common parent of both $c(v_k)$ and $c(v_{k+1})$.*

The following lemma shows that, when a worker p performs a tail-swap operation, if p's deque satisfies the contour property, then p's deque must have a specific structure.

Lemma 44 *At any time during an execution of a pipeline program which produces a computation dag $G = (V, E)$, suppose that worker p enables a vertex x via a throttling edge as a result of executing its assigned vertex v_0, which is the terminal of iteration i of some* `pipe_while` *loop. If p's deque satisfies the contour property (Definition 3), then one of the following conditions holds:*

1. *Worker p's deque is empty and x becomes p's new assigned vertex.*
2. *Worker p's deque contains a single vertex v_1, where the root of $c(v_1)$ is the root of iteration $i+1$ of the same* `pipe_while` *loop, and v_1 becomes p's new assigned vertex while x is pushed onto p's deque.*

Proof. Let $\langle v_0, v_1, \ldots, v_r\rangle$ denote the vertices in p's extended deque at the time v_0 is executed. If $r = 0$, then x becomes p's assigned vertex, satisfying Case 1 of the lemma. Otherwise, we have $r \ge 1$, and we shall show that, in fact, $r = 1$ and the root of $c(v_1)$ is the root of iteration $i+1$. In this situation, Case 2 of the lemma is satisfied as follows: because x is enabled by a throttling edge, a tail swap occurs, making v_1 the assigned vertex of p and pushing x onto p's deque.

Now we will show that, if $r \ge 1$, then we must have $r = 1$ and the root of $c(v_1)$ is the root of iteration $i+1$. Because x is enabled by a throttling edge, v_0 must be the terminal of some iteration i, and Lemma 43 implies that $c(x)$ is the parent of $c(v_0)$. The contour property (Definition 3) applied to v_0 states that either $c(v_1) = c(x)$ or $c(v_1)$ is the root of iteration $i+1$. The first case, $c(v_1) = c(x)$, is impossible, because Lemma 42 implies that vertices v_1 and x cannot simultaneously inhabit p's deque. We therefore have that $c(v_1)$ is the root of iteration $i+1$. In this case, the contour property and Lemma 42 imply that $r = 1$ as follows: if $r \geq 2$, then the contour property implies that $c(v_2) = c(x)$, and Lemma 42 implies that v_2 and x cannot simultaneously inhabit p's deque. □

Sections 7.7 and 7.8 use the structure of the workers' deques to bound the time and space, respectively, that Piper uses to execute a pipeline program. While Section 7.8 uses contours directly, Section 7.7 uses a property that Piper maintains while upholding the contour property. Intuitively, the analysis in Section 7.7 uses a measurement of the "distance" of each vertex in a worker's deque from the final vertex in the computation dag. To formalize this intuition, for a computation dag $G = (V, E)$, we define the ***enabling tree*** $G_T = (V, E_T)$ as the tree containing an edge $(u, v) \in E_T$ if u is the last predecessor of v to execute. The ***enabling depth*** $d(u)$ of $u \in V$ is the depth of u in the enabling tree G_T. Section 7.7 performs its analysis using the following "depth property," which roughly states that, except perhaps for the topmost vertex, the vertices in a worker's deque are sorted from bottom to top in order of decreasing depth:

Definition 4 *At any time during an execution of a pipeline program which produces a computation dag $G = (V, E)$, consider the extended deque $\langle v_0, v_1, \ldots, v_r \rangle$ of a worker p. This deque satisfies the* ***depth property*** *if the following conditions hold:*

1. *For $k = 1, 2, \ldots, r-1$, we have $d(v_{k-1}) \geq d(v_k)$.*
2. *For $k = r$, we have $d(v_{k-1}) \geq d(v_k)$ or v_k has an incoming throttling edge.*
3. *The inequalities are strict for $k > 1$.*

The depth property handles the topmost vertex in a worker's deque as a special case because the tail swap operation impedes our ability to relate the depth of this vertex to the depths of the other vertices in the deque. Although this special case prevents us from analyzing Piper by simply applying the analysis of Arora et al. [24], Section 7.7 extends the analysis of Arora et al. to overcome this hurdle.

The following theorem shows that, during the execution of a pipeline program by Piper, all workers' extended deques satisfy both the contour and the depth properties.

Theorem 45 *At all times during an execution of a pipeline program by* Piper*, all extended deques satisfy the contour and depth properties (Definitions 3 and 4).*

Proof. The proof follows a similar induction to the proof of Lemma 3 from [24]. Intuitively, we replace the "designated parents" discussed in [24] with contours, which exhibit similar parent-child relationships.

The claim holds vacuously in the base case, which is any empty deque.

Assuming inductively that the statement is true, consider the possible actions of Piper that modify the contents of the deque. For $r \geq 1$, let $v_0, v_1, \ldots, v_r$ denote the vertices on p's extended deque before p executes v_0, and let $v'_0, v'_1, \ldots, v'_{r'}$ denote the vertices on p's extended deque afterwards. Worker p can execute its assigned vertex v_0, thereby enabling 0, 1, or 2 vertices, or another worker q can steal a vertex from the top of the deque.

Worker q steals a vertex from p's deque. The statement holds because the identities of the remaining vertices in p's deque are unchanged. Similarly, the claim holds vacuously for q because q's extended deque has only the stolen vertex.

Executing v_0 enables 0 vertices. Worker p pops v_1 from the bottom of its deque to become its new assigned vertex v'_0. This action shifts all vertices in the deque down, that is, $r' = r - 1$ and for all k where $0 \leq k \leq r'$ we have $v'_k = v_{k+1}$. The statement holds because the identities of the remaining vertices in p's deque are unchanged.

Executing v_0 enables 1 vertex u. Worker p changes its assigned vertex from v_0 to $v'_0 = u$ and leaves all other vertices in the deque unchanged, that is, $r' = r$ and $v'_k = v_k$ for all $k > 1$. For vertices $v_2, v_3, \ldots, v_r$, if they exist, both Definitions 3 and 4 hold by induction. We therefore only need to consider the relationship between u and v_1.

The contour property holds by induction if $c(u) = c(v_0)$, that is, if the edge (v_0, u) is a serial or continue edge. The depth property also holds by induction because we are replacing v_0 on the extended deque with a successor node u, and thus $d(u) > d(v_0)$. Consequently, we need only consider the cases where (v_0, u) is either a spawn edge, a return edge, a cross edge, or a throttling edge.

- Edge (v_0, u) cannot be a spawn edge because executing a spawn always enables 2 children.
- If (v_0, u) is a return edge, then $c(u)$ is the parent of $c(v_0)$. In this case, we can show that p's deque is empty, which implies that the properties hold vacuously. For the sake of contradiction, suppose that p's deque contains a vertex v_1. By the inductive hypothesis, we have two cases:
  1. Contour $c(v_1)$ is the parent of contour $c(v_0)$. In this case, we have $c(v_1) = c(u)$. Lemma 42 tells us, however, that u and v_1 cannot simultaneously inhabit p's deque. Therefore, v_1 cannot exist in p's deque.
  2. The root of $c(v_0)$ is the root of some iteration i of a `pipe_while` loop and the root of $c(v_0)$ is the root of some iteration $i + 1$ of that loop. In this case, u terminates the `pipe_while` loop. One of u's predecessors, however, is the terminal vertex $s_{i+1,end}$ of iteration $i+1$, which vertex v_1 precedes in the dag. Vertex v_1 therefore cannot exist in p's deque.
- If (v_0, u) is a throttling edge, then Lemma 44 specifies the structure of worker p's extended deque. In particular, Lemma 44 states that the deque contains at most 1 vertex. If $r = 0$ then the deque is empty and the properties hold vacuously. Otherwise $r = 1$ and the deque contains one element v_1, in which case the tail-swap operation assigns v_1 to p and puts u into p's deque. The contour property holds, because $c(u)$ is the parent of $c(v_1)$. The depth property holds, because z is enabled by a throttling edge.
- Suppose that (v_0, u) is a cross edge. Lemma 43 shows that a cross edge (v_0, u) can only exist between vertices in sibling iteration contours. By the inductive hypothesis, $c(v_1)$ must be either the parent of $c(v_0)$ or equal to $c(u)$. In the latter case, however, enabling u would place two vertices from $c(u)$ on the same deque, which Lemma 42 says is impossible. Contour $c(v_1)$ is therefore the common parent of $c(v_0)$ and $c(u)$, and thus setting $v'_0 = u$ maintains the contour property. The depth property holds because u is a successor of v_0, and $d(u) > d(v_0)$.

Executing v_0 enables 2 vertices, u and w. Without loss of generality, assume that PIPER pushes the vertex w onto the bottom of its deque and assigns itself vertex u. Hence, we have $r' = r + 1$, vertex $v'_0 = u$, vertex $v'_1 = w$, and vertex $v'_k = v_{k-1}$ for all $1 < k \leq r'$. Definition 4 holds by induction, because the enabling edges (v_0, u) and (v_0, w) imply that $d(v_0) < d(u) = d(w)$. For vertices $v_2, v_3, \ldots, v_r$, if they exist, Definition 3 holds by induction. We therefore need only verify Definition 3 for vertices u and w.

To enable 2 vertices, v_0 must have at least 2 outgoing edges. We therefore have only three cases for v_0: either v_0 executes a `cilk_spawn`, or v_0 is the terminal of some iteration i in a `pipe_while` loop, or v_0 is a stage-counter vertex.

- If v_0 executes a `cilk_spawn`, then $c(w) = c(v_0)$ and $c(u)$ is a child contour of $c(v_0)$, maintaining Definition 3.
- If v_0 is the terminal of an iteration i, then it might have up to 3 outgoing edges: a cross edge (v_0, a), a throttling edge (v_0, x), and a return edge (v_0, z). We first observe that at most one of x or z is enabled, because both x and z belong to the same contour — the parent contour of $c(v_0)$ — and Lemma 42 therefore precludes enabling both x and z simultaneously. Consequently, u is the destination a of the cross edge and w is one of x or z.
  We now justify that the new contents of the deque satisfy Definition 3. By Lemma 43, we have that $c(v_0)$ and $c(u)$ are sibling contours corresponding to the adjacent iterations i and $i+1$, respectively, of a `pipe_while` loop, and $c(w)$ is the parent of both $c(v_0)$ and $c(u)$. Furthermore, we can show as follows that p's deque is otherwise empty. Lemma 44 specifies that the deque contains at most one vertex v_1, where the root of $c(v_1)$ is the root of iteration $i+1$. We therefore have $c(u) = c(v_1)$, and therefore, Lemma 42 states that u and v_1 cannot simultaneously appear on p's deque. Because executing v_0 enabled u, vertex v_1 cannot exist on p's deque, implying that p's deque is empty.
- If v_0 is a stage-counter vertex that is not the terminal of an iteration, then w must be the destination of a cross edge. In this case, vertex u is a node root in the same contour $c(u) = c(v_0)$, and by Lemma 43, $c(u)$ and $c(w)$ are adjacent siblings in the contour tree. As such, we need only show that $c(v_1)$, if it exists, is their parent. Suppose that $c(v_1)$ is not the parent of $c(v_0) = c(u)$, for the sake of contradiction. Then, by induction, it must be that $c(u)$ and $c(v_1)$ are adjacent siblings. We therefore have that $c(v_1) = c(w)$, which Lemma 42 shows is impossible.

□

7.7 Time analysis of Piper

This section bounds the completion time for Piper, showing that Piper executes pipeline program asymptotically efficiently. Specifically, suppose that a pipeline program produces a computation dag $G = (V, E)$ with work T_1 and span T_∞ when executed by Piper on P processors. We show that for any $\epsilon > 0$ the running time is $T_P \leq T_1/P + O(T_\infty + \lg P + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$, which implies that the expected running time is $T_P \leq T_1/P + O(T_\infty)$. This bound is comparable to the work-stealing bound for fork-join dags originally proved in [58].

We adapt the potential-function argument of Arora *et al.* [24], because Piper executes computation dags in a style similar to the their work-stealing scheduler, except for tail swapping. Although Arora *et al.* ignore the issue of memory contention, we handle it using the "recycling game" analysis of Blumofe and Leiserson [58], which contributes the additive $O(\lg P)$ term to the high-probability bounds.

The crux of the analysis is to bound the number of steal attempts performed during the execution of a computation dag in terms of its span. Following the analysis of Arora *et al.*, we measure progress through the computation by defining a potential function for a vertex in the computation dag based on its depth in the enabling tree. Consider a particular execution of a computation dag $G = (V, E)$ with span T_∞ by Piper. For that execution, we define the ***weight*** of a vertex v as $w(v) = T_\infty - d(v)$, and we define the ***potential*** of vertex v at a given time as

$$\phi(v) = \begin{cases} 3^{2w(v)-1} & \text{if } v \text{ is assigned ,} \\ 3^{2w(v)} & \text{otherwise .} \end{cases}$$

We define the potential of a worker p's extended deque $\langle v_0, v_1, \ldots, v_r \rangle$ as $\phi(p) = \sum_{k=0}^{r} \phi(v_k)$. The total potential of a computation at a given time is simply the sum of the potentials of each worker's deque. Arora *et al.* show that every action of their scheduler decreases the total potential over the course of the program's execution. We likewise use decreases in the total potential to measure the progress PIPER makes in executing a computation.

Given this potential function, the proof of the time bound follows the same overall structure as the analysis of Arora *et al.*. Conceptually, their analysis relies on the fact that the topmost node in each worker's deque has the smallest depth of all of the nodes in the deque, and therefore it accounts for a constant fraction of the total potential of the deque. Arora *et al.* use this property to argue that, after P steal attempts, with high probability, the topmost vertex in a particular deque is being executed. Consequently, the deque's potential has dropped by a constant fraction and progress therefore has been made.

Our potential-function analysis differs from that of Arora *et al.* because of the addition of the tail-swap operation in PIPER. First, we show that the potential of a worker's deque decreases when a tail-swap occurs. Second, as the depth property (Definition 4) shows, the tail-swap causes the topmost vertex in each worker's deque to not necessarily have the smallest depth. Consequently, this topmost vertex does not necessarily account for a constant fraction of the deque's total potential, meaning that the analysis of Arora *et al.* does not directly apply to PIPER. Conceptually, our analysis overcomes this hurdle by arguing that the top two vertices in each deque account for a constant fraction of the deque's potential. Hence, after $2P$ steal attempts, with high probability, a given deque's potential has probably decreased by a constant fraction, implying that progress has been made. The following lemmas and theorem formalize this conceptual approach.

First, we prove that a constant fraction of the potential of a worker's deque lies in its top two vertices and that every action of PIPER on a deque decreases that deque's potential.

Lemma 46 *At any time during an execution of a pipeline program which produces a computation dag $G = (V, E)$, the extended deque $\langle v_0, v_1, \ldots, v_r \rangle$ of every worker p satisfies the following properties:*

1. $\phi(v_r) + \phi(v_{r-1}) \geq 3\phi(p)/4$.
2. *Let ϕ' denote the potential after p executes v_0. Then we have*

$$\phi(p) - \phi'(p) = 2\left(\phi(v_0) + \phi(v_1)\right)/3 \ ,$$

if p performs a tail swap, and

$$\phi(p) - \phi'(p) \geq 5\phi(v_0)/9$$

otherwise.

PROOF. The analysis to show Property 1 is analogous to the analysis of Arora *et al.* [24, Lemma 6]. Because the result holds trivially for $r \leq 1$, we focus on the case where $r \geq 2$.

Because Theorem 45 shows that p's extended deque satisfies the depth property, we have

$$d(v_0) \geq d(v_1) > d(v_2) > \cdots > d(v_{r-2}) > d(v_{r-1}) \ .$$

The definition of the weight of a vertex, $w(v) = T_\infty - d(v)$, therefore gives us

$$w(v_0) \leq w(v_1) < w(v_2) < \cdots < w(v_{r-2}) < w(v_{r-1}) \ .$$

Because all weights are integers, we have that $w(v_{k-1}) \leq w(v_k) - 1$ for $k = 2, 3, \ldots, r-1$. Equivalently, we can bound all $w(v_k)$ in terms of $w(v_{r-1})$ as

$$w(v_k) \leq \begin{cases} w(v_{r-1}) - (r-1-k) & \text{if } 1 \leq k \leq (r-1) \ , \\ w(v_{r-1}) - (r-2) & \text{if } k = 0 \ . \end{cases}$$

For $r \geq 2$, we therefore have

$$\begin{aligned} \phi(p) &= \sum_{k=1}^{r} 3^{2w(v_k)} + 3^{2w(v_0)-1} \\ &= 3^{2w(v_r)} + 3^{2w(v_{r-1})} + \sum_{k=1}^{r-2} 3^{2w(v_k)} + \frac{1}{3} \cdot 3^{2w(v_0)} \\ &\leq 3^{2w(v_r)} + 3^{2w(v_{r-1})} + \left(\sum_{k=1}^{r-2} \frac{1}{3^{2(r-k-1)}} \right) \cdot 3^{2w(v_{r-1})} + \frac{1}{3} \cdot \frac{1}{3^{2(r-2)}} \cdot 3^{2w(v_{r-1})} \\ &= 3^{2w(v_r)} + 3^{2w(v_{r-1})} + \left(\frac{1}{8} + \frac{5}{24 \cdot 3^{2(r-2)}} \right) \cdot 3^{2w(v_{r-1})} \\ &\leq 3^{2w(v_r)} + 3^{2w(v_{r-1})} + \frac{1}{3} \cdot 3^{2w(v_{r-1})} \\ &= 3^{2w(v_r)} + \frac{4}{3} \cdot 3^{2w(v_{r-1})} \\ &\leq \frac{4}{3} \cdot \left(3^{2w(v_r)} + 3^{2w(v_{r-1})} \right) \ , \end{aligned}$$

and thus $\phi(v_r) + \phi(v_{r-1}) \geq 3\phi(p)/4$.

Now we argue that, in any time step t during which worker p executes its assigned vertex v_0, the potential of p's extended deque decreases. Let ϕ' denote the potential after the time step. If v_0 is the terminal of an iteration i and PIPER performs a tail swap after executing v_0, then Lemma 44 dictates the state of the deque before and after p executes v_0, from which we deduce that $\phi(p) - \phi'(p) = 2\left(\phi(v_0) + \phi(v_1)\right)/3$. The remaining cases follow from the analysis of Arora *et al.* [24], which shows that $\phi(p) - \phi'(p) \geq 5\phi(v_0)/9$. □

Similarly to Arora *et al.*, we analyze the behavior of workers randomly stealing from each other using a balls-and-weighted-bins analog. We want to analyze the case where the top 2 elements are stolen out of any deque, however, not just the top element. To address this case, we modify Lemma 7 of [24] to consider the probability that 2 out of $2P$ balls land in the same bin.

Lemma 47 *Consider P bins, where for $p = 1, 2, \ldots, P$, bin p has weight W_p. Suppose that $2P$ balls are thrown independently and uniformly at random into the P bins. For bin p,* *define the random variable X_p as*

$$X_p = \begin{cases} W_p & \text{if at least 2 balls land in bin } p \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

Let $W = \sum_{p=1}^{P} W_p$ and $X = \sum_{p=1}^{P} X_p$. For any β in the range $0 < \beta < 1$, we have $\Pr\{X \geq \beta W\} > 1 - 3/(1-\beta)e^2$.

Proof. For each bin p, consider the random variable $W_p - X_p$. It takes on the value W_p when 0 or 1 ball lands in bin p, and otherwise it is 0. Thus, we have

$$\begin{aligned} \mathrm{E}\left[W_p - X_p\right] &= W_p \left(\left(1 - \frac{1}{P}\right)^{2P} + 2P\left(1 - \frac{1}{P}\right)^{2P-1}\left(\frac{1}{P}\right) \right) \\ &= W_p \left(1 - \frac{1}{P}\right)^{2P} \frac{(3P-1)}{(P-1)} \ . \end{aligned}$$

Since $(1 - 1/P)^P$ approaches $1/e$ and $(3P-1)/(P-1)$ approaches 3, we have

$$\lim_{P \to \infty} \mathrm{E}\left[W_p - X_p\right] = 3W_p/e^2 \ .$$

In fact, one can show that $\mathrm{E}\left[W_p - X_p\right]$ is monotonically increasing, approaching the limit from below, and thus $\mathrm{E}\left[W - X\right] \leq 3W/e^2$. By Markov's inequality, we have that

$$\Pr\{(W - X) > (1-\beta)W\} < \mathrm{E}\left[W - X\right]/(1-\beta)W \ ,$$

from which we conclude that $\Pr\{X < \beta W\} \leq 3/(1-\beta)e^2$. □

To use Lemma 47 to analyze Piper, we divide the time steps of the execution of G into a sequence of ***rounds***, where each round (except the first, which starts at time 0) starts at the time step after the previous round ends and continues until the first time step such that at least $2P$ steal attempts — and hence less than $3P$ steal attempts — occur within the round. The following lemma shows that a constant fraction of the total potential in all deques is lost in each round, thereby demonstrating progress.

Lemma 48 *Consider the execution of a pipeline program by* Piper *on P processors. Suppose that a round starts at time step t and finishes at time step t'. Let ϕ denote the potential at time t, let ϕ' denote the potential at time t', let $\Phi = \sum_{p=1}^{P} \phi(p)$, and let $\Phi' = \sum_{p=1}^{P} \phi'(p)$. Then we have* $\Pr\{\Phi - \Phi' \geq \Phi/4\} > 1 - 6/e^2$.

Proof. We first show that stealing twice from a worker p's deque contributes a potential drop of at least $\phi(p)/2$. The proof follows a similar case analysis to that in the proof of Lemma 8 in [24] with two main differences. First, we use the two properties of ϕ in Lemma 46. Second, we must consider the case unique to Piper, where p performs a tail swap after executing its assigned vertex v_0.

We first observe that, if p is the target of at least 2 steal attempts, then Piper's actions on p's extended deque between time steps t and t' contribute a potential drop of at least $\phi(p)/2$. Let $\langle v_0, v_1, \ldots, v_r \rangle$ denote the vertices on p's extended deque at time t, and suppose that at least 2 steal attempts target p between time step t and time step t'.

- If p's extended deque is empty, then $\phi(p) = 0$, and the statement holds trivially.

- If $r = 0$, then p's extended deque consists solely of a vertex v_0 assigned to p, and $\phi(p) = \phi(v_0)$. By time t', worker p has executed vertex v_0, and Property 2 in Lemma 46 shows that the potential decreases by at least $5\phi(p)/9 \geq \phi(p)/2$.
- Suppose that $r > 1$. By time t', both v_r and v_{r-1} have been removed from p's deque, either by being stolen or by being assigned to p. In either case, the overall potential decreases by more than $2\phi(v_r)/3 + 2\phi(v_{r-1})/3$ — the decrease in potential from simply assigning v_r and v_{r-1} — which is $2\left(\phi(v_r) + \phi(v_{r-1})\right)/3 \geq 2\left(3\phi(p)/4\right)/3 = \phi(p)/2$ by Lemma 46.
- Suppose that $r = 1$. If executing v_0 causes p to perform a tail swap, then by Lemma 46, the potential drops by at least $2\left(\phi(v_0) + \phi(v_1)\right)/3 \geq \phi(p)/2$, since $\phi(p) = \phi(v_0) + \phi(v_1)$. Otherwise, by Lemma 46, the execution of v_0 results in an overall decrease in potential of $5\phi(v_0)/9$, and because p is the target of at least 2 steal attempts, v_1 is assigned by time t', decreasing the potential by $2\phi(v_1)/3$.

We now consider all P workers and $2P$ steal attempts between time steps t and t'. We model these steal attempts as ball tosses in the experiment described in Lemma 47. Suppose that we assign each worker p a weight of $W_p = \phi(p)/2$. These weights W_p sum to $W = \Phi/2$. If we think of steal attempts as ball tosses, then the random variable X_p from Lemma 47 bounds from below the potential decrease due to actions on p's deque. Specifically, if at least 2 steal attempts target p's deque in a round (which corresponds conceptually to at least 2 balls landing in bin p), then the potential drops by at least W_p. Moreover, X is a lower bound on the potential decrease within the round, that is, $X \leq \Phi - \Phi'$. By Lemma 47, we have $\Pr\{X \geq W/2\} > 1 - 6/e^2$. Substituting for X and W, we conclude that $\Pr\{(\Phi - \Phi') \geq \Phi/4\} > 1 - 6/e^2$. □

We are now ready to prove the completion-time bound.

Theorem 49 *Consider an execution of a pipeline program by* PIPER *on P processors which produces a computation dag with work T_1 and span T_∞. Then the expected running time is $T_P \leq T_1/P + O(T_\infty)$, and for any $\epsilon > 0$, the running time is $T_P \leq T_1/P + O(T_\infty + \lg P + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$.*

PROOF. On every time step, consider each worker as placing a token in a bucket depending on its action. If a worker p executes an assigned vertex, p places a token in the ***work bucket***. Otherwise, p is a thief and places a token in the ***steal bucket***. There are exactly T_1 tokens in the work bucket at the end of the computation. The interesting part is bounding the size of the steal bucket.

Divide the time steps of the execution of G into rounds. Recall that each round contains at least $2P$ and less than $3P$ steal attempts. Call a round ***successful*** if after that round finishes, the potential drops by at least a $1/4$ fraction. From Lemma 48, a round is successful with probability at least $1 - 6/e^2 \geq 1/6$. Since the potential starts at $\Phi_0 = 3^{2T_\infty - 1}$, ends at 0, and is always an integer, the number of successful rounds is at most $(2T_\infty - 1)\log_{4/3}(3) < 8T_\infty$. Consequently, the expected number of rounds needed to obtain $8T_\infty$ successful rounds is at most $48T_\infty$, and the expected number of tokens in the steal bucket is therefore at most $3P \cdot 48T_\infty = 144PT_\infty$.

For the high-probability bound, suppose that the execution takes $n = 48T_\infty + m$ rounds. Because each round succeeds with probability at least $p = 1/6$, the expected number of successes is at least $np = 8T_\infty + m/6$. We now compute the probability that the number X of successes is less than $8T_\infty$. As in [24], we use the Chernoff bound $\Pr\{X < np - a\} <$

$e^{-a^2/2np}$, with $a = m/6$. Choosing $m = 48T_\infty + 24\ln(1/\epsilon)$, we have

$$\Pr\{X < 8T_\infty\} < e^{\frac{-(m/6)^2}{16T_\infty + m/3}} < e^{\frac{-(m/6)^2}{m/3+m/3}} = e^{-m/24} \leq \epsilon \ .$$

Hence, the probability that the execution takes at least $n = 96T_\infty + 24\ln(1/\epsilon)$ rounds is less than ϵ, and the number of tokens in the steal bucket is at most $288T_\infty + 72\ln(1/\epsilon)$.

The "recycling game" analysis of Blumofe and Leiserson [58, Lemma 6] bounds the delay that might be incurred when multiple processors try to access the same deque in the same time step in randomized work stealing. This analysis adds $O(T_\infty)$ to the expected value of T_P and $O(T_\infty + \lg P + \lg(1/\epsilon))$ to the high-probability bound. □

7.8 Space analysis of Piper

This section derives bounds on the stack space required by Piper by extending the bounds of Blumofe and Leiserson [58] for fully strict fork-join parallelism to include pipeline parallelism. We show that Piper on P processors uses $S_P \leq P(S_1 + fDK)$ stack space for pipeline iterations, where S_1 is the serial stack space, f is the "frame size," D is the depth of nested linear pipelines, and K is the throttling limit.

To model Piper's usage of stack space, we partition the vertices of a pipeline computation dag G into a tree of contours, in a similar manner to that described in Section 7.6. We assume that every contour c of G has an associated ***frame size*** representing the stack space consumed by c while it or any of its descendant contours are executing. The space used by Piper on any time step is the sum of frame sizes of all contours c which are either ***active*** — c is associated with a vertex in some worker's extended deque — or ***suspended*** — the earliest unexecuted vertex in the contour is not ready.

As Section 7.4 describes, the contours of a `pipe_while` loop do not directly correspond to the control and iteration frames allocated for the loop. In particular, as demonstrated in the code transformation in Figure 7-5, stage 0 allocates an iteration frame for all stages of the iteration and executes using that iteration frame. To account for the space used when executing an iteration i of a `pipe_while` loop, consider an active or suspended contour c, and let v be either the vertex in c on a worker's deque, if c is active, or the earliest unexecuted vertex in c, if c is suspended. If v lies on a path in c between a stage 0 node root $a_{i,0}$ and its corresponding node terminal $b_{i,0}$ for some iteration i of a `pipe_while` loop, then v incurs an additional space cost equal to the size of i's iteration frame.

The following theorem bounds the stack space used by Piper. Let S_P denote the maximum over all time steps of the stack space Piper uses during a P-worker execution of G. Thus, S_1 is the stack space used by Piper for a serial execution. Define the ***pipe nesting depth*** D of G as the maximum number of `pipe_while` contours on any path from leaf to root in the contour tree. The following theorem generalizes the space bound $S_P \leq PS_1$ from [58], which deals only with fork-join parallelism, to pipeline programs.

Theorem 50 *Consider a pipeline program with pipe nesting depth D executed on P processors by* Piper *with throttling limit K. The execution requires $S_P \leq P(S_1 + fDK)$ stack space, where f is the maximum frame size of any contour of any* `pipe_while` *iteration and S_1 is the serial stack space.*

Proof. We show that, at each time step during Piper's execution of a pipeline program, Piper satisfies a variant of the "busy-leaves property" [58] with respect to the tree of active and suspended contours during that time step. This proof follows a similar induction to that in [58, Thoerem 3], with one change, namely, that a leaf contour c may stall if the earliest unexecuted vertex in c is the destination of a cross edge.

If v is the destination of a cross edge, then v must be associated with some `pipe_while` loop, specifically, as a node root $a_{i,j}$ for some iteration i in the loop. The source of this cross edge must be in a previous iteration of the same `pipe_while` loop. Consider the ***leftmost*** iteration $i' < i$ of this `pipe_while` loop, that is, the iteration with the smallest iteration index that has not completed. By definition of the leftmost iteration, all previous iterations in this `pipe_while` have completed, and thus no node root $a_{i',j}$ in iteration i' may be the destination of a cross edge whose source has not executed. In other words, the contour associated with this leftmost iteration must be active or have an active descendant. Consequently, each suspended contour c is associated with a `pipe_while` loop, and for each such contour c, there exists an active sibling contour associated with the same `pipe_while` loop.

We account for the stack space PIPER uses by separately considering the active and suspended leaf contours. Because there are P processors executing the computation, at each time step, PIPER has at most P active leaf contours, each of which may use at most S_1 stack space. Each suspended leaf contour, meanwhile, is associated with a `pipe_while` loop, whose leftmost iteration during this time step either is active or has an active descendant. Any `pipe_while` loop uses at most fK stack space to execute its iterations, because the throttling edge from the terminal of its leftmost iteration precludes having more than K active or suspended iterations in any one `pipe_while` loop. Thus, for each vertex in its deque that is associated with an iteration of a `pipe_while` loop, each worker p accounts for at most fK stack space in suspended sibling contours for iterations of the same `pipe_while` loop, or fDK stack space overall. Summing the stack space used over all workers gives $PfDK$ additional stack-space usage. $\square$

7.9 Cilk-P runtime design

This section describes the Cilk-P implementation of the PIPER scheduler. We first introduce the data structures Cilk-P uses to implement a `pipe_while` loop. Then we describe the two main optimizations that the Cilk-P runtime exploits: "lazy enabling" and "dynamic dependency folding."

Data structures

Like the Cilk-M runtime [238] on which it is based, Cilk-P organizes runtime data into frames. The transformed code in Figure 7-5 reflects the frames Cilk-P allocates to execute a `pipe_while` loop. In particular, Cilk-P executes a `pipe_while` loop in its own control frame, which handles the spawning and throttling of iterations. Furthermore, each iteration of a `pipe_while` loop executes as an independent child function with its own iteration frame. This frame structure is similar to that of an ordinary `while` loop in Cilk-M, where each iteration spawns a function to execute the loop body. Cross and throttling edges, however, may cause the iteration and control frames to suspend.

Cilk-P's runtime employs a simple mechanism to track progress of an iteration i. As seen in Figure 7-5, the frame of iteration i maintains a stage counter, which stores the stage number of the currently executing or suspended node in i. In addition, the iteration i's frame maintains a ***status*** field, which indicates whether i is suspended due to an unsatisfied cross edge. Because executed nodes in an iteration i have strictly increasing stage numbers, checking whether a cross edge into iteration i is satisfied amounts to comparing the stage counters of iterations i and $i-1$. Any iteration frame that is not suspended corresponds to either a currently executing or a completed iteration.

Cilk-P implements throttling using a ***join counter*** in the control frame. Normally in Cilk-M, a frame's join counter simply stores the number of active child frames. Cilk-P also uses the join counter to limit the number of active iteration frames in a `pipe_while` loop to the throttling limit K. Starting an iteration increments the join counter, while returning from the earliest active iteration decrements it. (For implementation simplicity, Cilk-P additionally ensures that iterations return in order.) If a worker tries to start a new iteration when the control frame's join counter is K, the control frame suspends until a child iteration returns.

Using these data structures, one could implement PIPER directly, by pushing and popping the appropriate frames onto deques as specified by PIPER's execution model. In particular, the normal THE protocol [146] in Cilk could be used for pushing and popping frames from a deque, and frame locks could be used to update fields in the frames atomically. Although this approach directly matches the model analyzed in Sections 7.7 and 7.8, it incurs unnecessary overhead for every node in an iteration. Cilk-P implements "lazy enabling" and "dynamic dependency folding" to reduce this overhead.

Lazy enabling

In the PIPER algorithm, when a worker p finishes executing a node in iteration i, it may enable an instruction in iteration $i+1$, in which case p pushes this instruction onto its deque. To implement this behavior, intuitively, p must ***check right*** — read the stage counter and status of iteration $i+1$ — whenever it finishes executing a node. The work to check right at the end of every node can amount to substantial overhead in a pipeline with fine-grained stages.

Lazy enabling allows p's execution of an iteration i to defer the check-right operation, as well as avoid any operations on its deque involving iteration $i+1$. Conceptually, when p enables work in iteration $i+1$, this work is kept on p's deque implicitly. When a thief q tries to steal iteration i's frame from p's deque, q first checks right on behalf of p to see whether any work from iteration $i+1$ is implicitly on the deque. If so, q resumes iteration $i+1$ as if it had found it on p's deque. In a similar vein, the Cilk-P runtime system also uses lazy enabling to optimize the ***check-parent*** operation — the enabling of a control frame suspended due to throttling.

Lazy enabling requires p to behave differently when p completes an iteration. When p finishes iteration i, it first checks right, and if that fails (because iteration $i+1$ need not be resumed), it checks its parent. It turns out that these checks find work only if p's deque is empty, that is, if all other work on p's deque has been stolen. Therefore, p can avoid performing these checks at the end of an iteration if its deque is not empty.

Lazy enabling is an application of the ***work-first principle*** [146]: minimize the scheduling overheads borne by the work of a computation, and amortize them against the span. Requiring a worker to check right every time it completes a node adds overhead proportional to the work of the `pipe_while` in the worst case. With lazy enabling, the overhead can be amortized against the span of the computation. For programs with sufficient parallelism, the work dominates the span, and the overhead becomes negligible.

CPU	AMD Opteron 8354
Clock	2 GHz
Cores per processor chip	4
Processor chips (sockets)	4
L1 data cache/core	64 KiB
L2 cache/core	512 KiB
L3 cache/socket	2 MiB
DRAM	128 GiB DDR3
Compiler	GCC (G++ for TBB) 4.4.5
Operating system	MIT CSAIL Debian 6.08 (squeeze), custom Linux kernel 3.4.0

Figure 7-8: Technical specifications of the AMD Opteron 8354 system used for one set of experiments. The Linux kernel was patched with support for thread-local memory mapping for Cilk-M [238].

Dynamic dependency folding

In ***dynamic dependency folding***, the frame for iteration i stores a cached value of the stage counter of iteration $i-1$, hoping to avoid checking already satisfied cross edges. In a straightforward implementation of PIPER, before a worker p executes each node in iteration i with an incoming cross edge, it reads the stage counter of iteration $i-1$ to see if the cross edge is satisfied. Reading the stage counter of iteration $i-1$, however, can be expensive. Besides the work involved, the access may contend with the worker q executing iteration $i-1$, because q might be frequently updating the stage counter of iteration $i-1$.

Dynamic dependency folding mitigates this overhead by exploiting the fact that an iteration's stage counter must strictly increase. By caching the most recently read stage-counter value from iteration $i-1$, worker p can sometimes avoid reading this stage counter before each node with an incoming cross edge. For instance, if q finishes executing a node $(i-1, j)$, then all cross edges from nodes $(i-1, 0)$ through $(i-1, j)$ are necessarily satisfied. Thus, if p reads j from iteration $i-1$'s stage counter, p need not reread the stage counter of $i-1$ until it tries to execute a node with an incoming cross edge (i, k) where $k > j$. This optimization is particularly useful for fine-grained stages that execute quickly.

7.10 Evaluation

This section presents empirical studies of the Cilk-P prototype system. We investigated the performance and scalability of Cilk-P using the three PARSEC [45, 46] benchmarks that we ported, namely *ferret*, *dedup*, and *x264*. The results show that Cilk-P's implementation of pipeline parallelism has negligible overhead compared to its serial counterpart. We compared the Cilk-P implementations to TBB and Pthreaded implementations of these benchmarks. We found that the Cilk-P and TBB implementations perform comparably, as do the Cilk-P and Pthreaded implementations for *ferret* and *x264*. The Pthreaded version of *dedup* outperforms both Cilk-P and TBB, because the bind-to-element approaches of Cilk-P and TBB produce less parallelism than the Pthreaded bind-to-stage approach. Moreover, the Pthreading approach benefits more from "oversubscription," that is, using more threads than the number of available hardware cores. We study the effectiveness of dynamic dependency folding on a synthetic benchmark called *pipe-fib*, demonstrating that this optimization can be effective for applications with fine-grained stages.

We ran two sets of experiments on two different machines. The first set was collected on the AMD Opteron system described in Figure 7-8. The second set was collected on

CPU	Intel Xeon E5-2665
Clock	2.4 GHz
Cores per processor chip	8
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	256 KiB
L3 cache/socket	20 MiB
DRAM	32 GiB DDR3
Compiler	GCC (G++ for TBB) 4.6.3
Operating system	Fedora 16, custom Linux kernel 3.6.11

Figure 7-9: Technical specifications of the Intel Xeon E5-2665 system used for one set of experiments. The Linux kernel was patched with support for thread-local memory mapping for Cilk-M [238].

the Intel Xeon system described in Figure 7-9. The benchmarks were compiled using `-O3` optimization, except for *x264*, which by default comes with `-O4`. The Intel Xeon machine is several years newer than the AMD machine, which precludes a direct comparison between these systems.

Performance evaluation on PARSEC benchmarks

We implemented the Cilk-P versions of the three PARSEC benchmarks by hand-compiling the relevant `pipe_while` loops using techniques similar to those described in [238]. We then compiled the hand-compiled benchmarks with GCC. The *ferret* and *dedup* applications can be parallelized as simple pipelines with a fixed number of stages and a static dependency structure. In particular, *ferret* uses the 3-stage SPS pipeline shown in Figure 7-1, while *dedup* uses a 4-stage SSPS pipeline as described in Figure 7-4.

For the Pthreaded versions, we used the code distributed with PARSEC. The PARSEC Pthreaded implementations of the *ferret* and *dedup* benchmarks employ the ***oversubscription method*** [328], a bind-to-stage approach that creates more than one thread per pipeline stage and utilizes the operating system for load balancing. For the Pthreaded implementations, when the user specifies an input parameter of Q, the code creates Q threads per stage, except for the first (input) and last (output) stages which are serial and use only one thread each. To ensure a fair comparison, for all applications, we ran the Pthreaded implementation using `taskset` to limit the process to P cores (which corresponds to the number of workers used in Cilk-P and TBB), but experimented to find the best setting for Q.

We used the TBB version of *ferret* that came with the PARSEC benchmark, and we implemented the TBB version of *dedup*. Both TBB implementations use the same pipelining strategies as their Cilk-P counterparts. TBB's construct-and-run approach proved inadequate for the on-the-fly nature of *x264*, however, and indeed, in their study of these three applications, Reed, Chen, and Johnson [328] say, "Implementing *x264* in TBB is not impossible, but the TBB pipeline structure is not suitable." Thus, we had no TBB benchmark for *x264* to include in our comparisons.

For each benchmark, we throttled all versions similarly, unless specified otherwise in the figure captions. For Cilk-P on the AMD Opteron system, we used the default throttling limit of $4P$, where P is the number of cores. This default value seems to work well in general, although since *ferret* scales slightly better with less throttling, we used a throttling limit of $10P$ for *ferret* in our experiments. Similarly, for the Intel Xeon system, we used a throttling limit of $10P$ for our experiments, which seemed to work well. TBB supports

	Processing Time (T_P)			Speedup (T_S/T_P)			Scalability (T_1/T_P)		
P	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB
1	432.4	430.0	432.7	1.01	1.01	1.00	1.00	1.00	1.00
2	220.4	212.2	223.8	1.97	2.05	1.94	1.96	2.03	1.93
3	146.9	140.8	147.0	2.96	3.09	2.96	2.94	3.05	2.94
4	111.5	106.0	111.8	3.90	4.10	3.89	3.88	4.06	3.87
5	89.2	89.9	90.8	4.87	4.83	4.79	4.85	4.78	4.77
6	74.8	73.8	76.1	5.81	5.88	5.71	5.78	5.82	5.67
7	64.7	64.2	65.9	6.71	6.76	6.59	6.68	6.70	6.57
8	57.3	57.0	57.7	7.58	7.62	7.53	7.54	7.54	7.50
9	51.1	49.8	52.9	8.50	8.72	8.34	8.46	8.64	8.31
10	46.4	45.5	47.3	9.36	9.55	9.19	9.32	9.46	9.16
11	42.5	41.7	43.2	10.22	10.41	10.05	10.18	10.30	10.01
12	39.4	38.6	40.0	11.03	11.26	10.85	10.98	11.15	10.81
13	36.6	37.2	37.6	11.87	11.67	11.54	11.82	11.55	11.49
14	34.4	35.0	35.3	12.64	12.41	12.29	12.58	12.28	12.25
15	32.2	32.9	33.5	13.48	13.19	12.96	13.42	13.06	12.91
16	30.7	31.3	31.8	14.17	13.89	13.67	14.07	13.75	13.61

Figure 7-10: Performance comparison of the three *ferret* implementations running on the AMD Opteron system. The experiments were conducted using *native*, the largest input data set that comes with the PARSEC benchmark suite.[4] The left-most column shows the number of cores used (P). Subsequent columns show the running time (T_P), speedup (T_S/T_P) over serial running time $T_S = 434.4$ seconds, and scalability (T_1/T_P) for each system. The throttling limit was $K = 10P$.

a settable parameter that serves the same purpose as Cilk-P's throttling limit. For the Pthreaded implementations, we throttled the computation by setting a size limit on the queues between stages, although we did not impose a queue size limit on the last stage of *dedup* (the default limit is 2^{20}), since doing so causes the program to deadlock.

Figures 7-10 through 7-12 show the performance results for the different implementations of the three benchmarks running on the AMD Opteron system. Each data point in the study was computed by averaging the results of 10 runs. The standard deviation of the numbers was less than 5 % for a majority of the data points, except for a couple outliers for which the standard deviation was about 10 %. We suspect that the superlinear scalability obtained for some measurements is due to the fact that more L1- and L2-cache is available when running on multiple cores.

The tables from Figures 7-10 and 7-11 show that the Cilk-P and TBB implementations of *ferret* and *dedup* are comparable, indicating that there is no performance penalty incurred by these applications for using the more general on-the-fly pipeline instead of a construct-and-run pipeline. Recall that both Cilk-P and TBB execute using a bind-to-element approach.

The *dedup* performance results (Figure 7-11) for Cilk-P and TBB are inferior to those for Pthreads, however. The Pthreaded implementation scales to about 8.5 on 16 cores, whereas Cilk-P and TBB seem to plateau at around 6.7. There appear to be two reasons for this discrepancy.

First, the *dedup* benchmark on the test input has limited parallelism. We modified the Cilkview scalability analyzer [180] to measure the work and span of our hand-compiled Cilk-P *dedup* programs, and we measured the parallelism of *dedup* to be merely 7.4. The

[4]We dropped four out of the 3500 input images from the original *native* data set, because those images are black-and-white, which trigger an array index out of bound error in the image library provided.

	Processing Time (T_P)			Speedup (T_S/T_P)			Scalability (T_1/T_P)		
P	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB
1	56.4	51.7	55.9	1.04	1.13	1.05	1.00	1.00	1.00
2	29.5	23.5	29.5	1.99	2.49	1.98	1.91	2.20	1.89
3	20.1	15.8	20.4	2.91	3.71	2.88	2.80	3.28	2.75
4	15.8	12.4	16.1	3.71	4.73	3.64	3.57	4.17	3.47
5	13.5	11.3	13.7	4.33	5.19	4.28	4.16	4.58	4.09
6	11.9	10.5	12.1	4.92	5.56	4.85	4.73	4.90	4.63
7	10.8	9.5	11.0	5.42	6.18	5.33	5.22	5.45	5.09
8	10.1	8.6	10.2	5.82	6.81	5.73	5.61	6.01	5.48
9	9.5	7.6	9.6	6.15	7.69	6.09	5.92	6.79	5.81
10	9.1	7.1	9.2	6.42	8.28	6.35	6.18	7.31	6.07
11	8.8	6.8	9.0	6.65	8.61	6.54	6.40	7.59	6.24
12	8.6	6.7	8.8	6.80	8.75	6.67	6.54	7.72	6.37
13	8.5	6.7	8.7	6.93	8.80	6.76	6.67	7.76	6.46
14	8.3	6.5	8.6	7.04	9.04	6.79	6.78	7.98	6.49
15	8.3	6.2	8.6	7.08	9.48	6.84	6.82	8.37	6.54
16	8.2	6.0	8.5	7.12	9.76	6.88	6.85	8.61	6.57

Figure 7-11: Performance comparison of the three *dedup* implementations running on the AMD Opteron system. The experiments were conducted using *native*, the largest input data set that comes with the PARSEC benchmark suite. The column headers are the same as in Figure 7-10. The serial running time T_S was 58.6 seconds. The throttling limit was $K = 4P$.

bind-to-stage Pthreaded implementation creates a pipeline with a different structure from the bind-to-element Cilk-P and TBB versions, which enjoys slightly more parallelism.

Second, since file I/O is the main performance bottleneck for *dedup*, the Pthreaded implementation effectively benefits from oversubscription and its strategic allocation of threads to stages. Specifically, since the first and last stages perform file I/O, which is inherently serial, the Pthreaded implementation dedicates one thread to each of these stages, but dedicates multiple threads to the other compute-intensive stages. While the writing thread is performing file I/O to write data out to the disk, the OS can deschedule it, allowing the compute-intensive threads to be scheduled. This behavior explains how the Pthreaded implementation scales by more than a factor of P for $P = 1$–4, even though the computation is restricted to only P cores using `taskset`. Moreover, when we ran the Pthreaded implementation without throttling on a single core, the computation ran about 20 % faster than the original serial implementation of *dedup*. This performance boost might be explained by the computation and file I/O operations are effectively overlapped. Oversubscription with throttling, meanwhile, improves the performance of the Pthreaded implementation because throttling appears to inhibit threads working on stages that are further ahead, allowing threads working on heavier stages to obtain more processing resources, thereby balancing the load.

Figures 7-13 through 7-15 show the performance results for the different implementations of the three benchmarks running on the Intel Xeon system. For each benchmark, the relative performance between the three implementations follows similar trends as in the results for the AMD Opteron system.

The results from the two systems differ in the following ways. First, the serial running time across implementations for each benchmark is about 2–3 times faster on the Intel Xeon system than on the AMD Opteron system. This discrepancy of serial running times can be explained by the fact that the Intel Xeon processors have higher clock frequency and that the

	Encoding Time (T_P)		Speedup (T_S/T_P)		Scalability (T_1/T_P)	
P	Cilk-P	Pthreads	Cilk-P	Pthreads	Cilk-P	Pthreads
1	211.1	219.8	1.04	0.99	1.00	1.00
2	99.1	103.5	2.21	2.11	2.13	2.12
3	65.3	67.8	3.35	3.22	3.23	3.24
4	49.6	51.6	4.40	4.23	4.25	4.26
5	40.9	42.0	5.34	5.21	5.16	5.24
6	34.4	35.8	6.35	6.11	6.13	6.14
7	29.9	31.5	7.31	6.94	7.06	6.98
8	26.7	28.5	8.20	7.66	7.92	7.70
9	23.9	25.8	9.13	8.49	8.82	8.53
10	22.0	23.0	9.92	9.50	9.58	9.55
11	20.5	21.0	10.68	10.41	10.31	10.47
12	19.1	19.5	11.47	11.18	11.07	11.25
13	18.0	18.7	12.12	11.69	11.70	11.76
14	17.1	17.4	12.82	12.56	12.38	12.63
15	16.4	16.5	13.34	13.21	12.88	13.28
16	15.8	16.0	13.81	13.67	13.34	13.75

Figure 7-12: Performance comparison between the Cilk-P implementation and the Pthreaded implementation of *x264* (encoding only) running on the AMD Opteron system. The experiments were conducted using *native*, the largest input data set that comes with the PARSEC benchmark suite. The column headers are the same as in Figure 7-10. The serial running time T_S was 218.6 seconds. The throttling limit was $K = 4P$.

system overall has more memory. In addition, the memory bandwidth on the Intel system is about 2–3 times higher than on the AMD system[5], depending on the data size accessed by the computation. Second, as shown in Figure 7-14, we observed less speedup from *dedup* across the three implementations on the Intel system than on the AMD system. The reason for this reduced speedup is because the number of last-level cache misses is increased substantially on the Intel system between a serial execution and a parallel execution. This increase in number of cache misses causes the time spent in the user code to double when running on 16 processors compared with running serially. The AMD system, on the other hand, does not appear to exhibit the same cache behavior. We are unsure of what exactly creates this discrepancy in cache behavior between the two systems, but our measurements suggest that these additional cache misses come mostly from the compress library used by *dedup*, for which we lack the source code.

In summary, Cilk-P performs comparably to TBB while admitting more expressive semantics for pipelines. Cilk-P also performs comparably to the Pthreaded implementations of *ferret* and *x264*, although its bind-to-element strategy seems to suffer on *dedup* compared to the bind-to-stage strategy of the Pthreaded implementation. Despite losing the *dedup* "bake-off," Cilk-P's strategy has the significant advantage that it allows pipelines to be expressed as deterministic programs. In contrast, the Pthreaded pipelines are inherently nondeterministic, which complicates their debugging and maintenance.

Evaluation of dynamic dependency folding

We also studied the effectiveness of dynamic dependency folding. Since the PARSEC benchmarks are too coarse grained to permit such a study, we implemented a synthetic benchmark,

[5]We measured the memory latencies using version 3.0-a9 of lmbench [279].

	Processing Time (T_P)			Speedup (T_S/T_P)			Scalability (T_1/T_P)		
P	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB
1	153.2	152.5	151.5	1.04	1.04	1.05	1.00	1.00	1.00
2	77.5	89.7	77.0	2.05	1.77	2.06	1.98	1.70	1.97
3	51.8	56.9	53.5	3.07	2.79	2.97	2.96	2.68	2.83
4	39.9	42.8	40.0	3.99	3.72	3.98	3.84	3.57	3.79
5	32.3	34.4	32.5	4.93	4.63	4.89	4.75	4.44	4.66
6	27.4	29.7	27.6	5.81	5.36	5.75	5.59	5.14	5.48
7	23.5	25.7	24.0	6.76	6.18	6.61	6.51	5.93	6.30
8	21.0	22.7	21.3	7.57	7.00	7.45	7.29	6.72	7.10
9	19.0	21.8	19.4	8.37	7.31	8.20	8.07	7.01	7.81
10	17.3	18.8	17.8	9.19	8.46	8.94	8.86	8.11	8.52
11	15.8	17.3	16.4	10.05	9.20	9.68	9.68	8.82	9.23
12	14.7	15.8	15.3	10.81	10.08	10.37	10.42	9.67	9.88
13	13.8	14.6	14.5	11.52	10.88	10.98	11.10	10.43	10.47
14	13.0	13.7	13.8	12.20	11.64	11.56	11.76	11.17	11.02
15	12.2	12.9	13.1	12.98	12.30	12.15	12.51	11.80	11.58
16	11.6	12.2	12.5	13.77	13.02	12.70	13.26	12.48	12.10

Figure 7-13: Performance comparison of the three *ferret* implementations running on the Intel Xeon system. The experiments were conducted using *native*, and the column headers are the same as in Figure 7-10. The serial running time T_S was 159.0 seconds. The throttling limit was $K = 10P$.

called *pipe-fib*, to study this optimization technique. The *pipe-fib* benchmark computes the nth Fibonacci number F_n in binary. It uses a pipeline algorithm that operates in $\Theta(n^2)$ work and $\Theta(n)$ span. To construct the base case, *pipe-fib* allocates three arrays of size $\Theta(n)$ and initializes the first two arrays with the binary representations of F_1 and F_2, both of which are 1. To compute F_3, *pipe-fib* performs a ripple-carry addition on the two input arrays and stores the sum into the third output array. To compute F_n, *pipe-fib* repeats the addition by rotating through the arrays for inputs and output until it reaches F_n. In the pipeline for this computation, each iteration i computes F_{i+2}, and a stage j within the iteration computes the jth bit of F_{i+2}. Since the benchmark stops propagating the carry bit as soon as possible, it generates a triangular pipeline dag in which the number of stages increases with iteration number. Given that each stage in *pipe-fib* starts with a `pipe_stage_wait`, and each stage contains little work, it serves as an excellent microbenchmark to study the overhead of `pipe_stage_wait`.

Figure 7-16 shows the performance results[6] on the AMD Opteron system, obtained by running the ordinary *pipe-fib* with fine-grained stages, as well as *pipe-fib-256*, a coarsened version of *pipe-fib* in which each stage computes 256 bits instead of 1. As the data in the first row show, even though the serial overhead for *pipe-fib* without coarsening is merely 13 %, it fails to scale and exhibits poor speedup. The reason is that checking for dependencies due to cross edges has a relatively high overhead compared to the little work in each fine-grained stage. As the data for *pipe-fib-256* in the second row show, coarsening the stages improves both serial overhead and scalability. Ideally, one would like the system to coarsen automatically, which is what dynamic dependency folding effectively achieves.

Further investigation revealed that the time spent checking for cross edges increases noticeably when the number of workers increases from 1 to 2. It turns out that when iterations are run in parallel, each check for a cross-edge dependency necessarily incurs a ***true-sharing***

[6]Figure 7-16 shows the results from runs with a single input size, but these data are representative of other runs with different input sizes.

	Processing Time (T_P)			Speedup (T_S/T_P)			Scalability (T_1/T_P)		
P	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB
1	29.6	29.2	30.1	1.01	1.02	0.99	1.00	1.00	1.00
2	18.0	14.6	21.1	1.65	2.03	1.41	1.64	2.00	1.43
3	12.8	10.0	18.6	2.32	2.96	1.60	2.30	2.92	1.62
4	10.4	8.2	15.7	2.85	3.62	1.89	2.83	3.56	1.92
5	9.3	7.5	15.4	3.20	3.95	1.94	3.19	3.89	1.96
6	8.5	8.0	14.5	3.48	3.71	2.05	3.46	3.65	2.08
7	8.0	7.2	13.7	3.70	4.12	2.17	3.68	4.05	2.20
8	7.6	6.5	13.9	3.89	4.55	2.13	3.86	4.48	2.16
9	7.5	7.2	13.3	3.94	4.13	2.24	3.92	4.06	2.27
10	7.5	6.7	12.5	3.95	4.42	2.38	3.93	4.35	2.41
11	7.6	7.1	12.1	3.91	4.17	2.45	3.89	4.10	2.49
12	7.8	7.3	11.2	3.81	4.05	2.65	3.79	3.98	2.69
13	8.1	7.5	11.1	3.67	3.97	2.68	3.65	3.90	2.72
14	8.5	7.8	11.0	3.50	3.84	2.70	3.48	3.77	2.74
15	8.8	8.8	10.4	3.38	3.36	2.85	3.36	3.30	2.89
16	9.3	9.1	10.4	3.21	3.26	2.86	3.19	3.20	2.90

Figure 7-14: Performance comparison of the three *dedup* implementations running on the Intel Xeon system. The experiments were conducted using *native*, and the column headers are the same as in Figure 7-10. The serial running time T_S was 29.7 seconds. The throttling limit was $K = 10P$ for Cilk-P and TBB, and $4P$ for Pthreads, because the Cilk-P and TBB implementations performed better with throttling limit of $10P$ than $4P$, whereas the Pthreaded implementation was the other way around.

conflict between the two adjacent active iterations, in which parallel workers simultaneously access the same shared memory location. Dynamic dependency folding eliminated much of this overhead for *pipe-fib*, as shown in the third row of Figure 7-16, leading to scalability that is much closer to the coarsened version without the optimization, although a slight price is still paid in speedup. Employing both optimizations, as shown in the last row of the table, produces the best numbers for both speedup and scalability.

We have done a similar performance study on the Intel Xeon system, and the relative performance of *pipe-fib* and *pipe-fib-256*, with and without dependency folding, show similar trends as in the results on the AMD system. The parallel running times on the Intel system are affected more by the cache misses due to ***false sharing*** on the three arrays, in which parallel workers access different locations in these arrays that happen to lie in the same cache line. This false sharing causes dependency folding to produce less speedup for *pipe-fib* on the Intel system, specifically, a speedup of 8.8. This false-sharing effect disappears if *pipe-fib* is modified to employ a larger data type for the array and is coarsened slightly.

In this section, we have empirically evaluated the performance and scalability of the Cilk-P prototype system. Provided that the application has ample parallelism, such as in the case of *ferret* and *x264*, Cilk-P demonstrates good scalability on machines that we tested, which each contain 16 cores. One might wonder, can one expect Cilk-P to continue to scale as future multicore systems contain increasingly more cores? Since Cilk-P implements Piper, its provably good time bound predicts near-linear speedup assuming the application contains ample parallelism. Due to scheduling overhead, however, one would not expect an application to scale linearly up to T_1/T_∞ processors, even though theory predicts that the application could use that many processors. The question then becomes how much parallelism is considered ample for the application to scale linearly with respect to the

	Encoding Time (T_P)		Speedup (T_S/T_P)		Scalability (T_1/T_P)	
P	Cilk-P	Pthreads	Cilk-P	Pthreads	Cilk-P	Pthreads
1	95.1	95.2	1.02	1.02	1.00	1.00
2	49.4	51.9	1.97	1.87	1.93	1.83
3	33.4	35.5	2.91	2.73	2.85	2.68
4	25.5	27.9	3.82	3.48	3.74	3.41
5	20.8	22.6	4.66	4.30	4.56	4.21
6	17.5	19.6	5.53	4.97	5.42	4.86
7	15.3	16.9	6.36	5.74	6.22	5.62
8	13.4	14.9	7.24	6.50	7.08	6.37
9	12.2	13.4	7.97	7.26	7.80	7.11
10	11.1	12.1	8.73	8.01	8.55	7.84
11	10.4	11.0	9.37	8.83	9.17	8.65
12	9.6	10.2	10.07	9.54	9.85	9.34
13	9.2	9.5	10.55	10.21	10.32	10.00
14	8.8	9.0	11.04	10.80	10.80	10.58
15	8.3	8.6	11.68	11.34	11.43	11.12
16	8.0	8.3	12.08	11.77	11.82	11.53

Figure 7-15: Performance comparison between the Cilk-P implementation and the Pthreaded implementation of *x264* (encoding only) running on the Intel Xeon system. The experiments were conducted using *native*, and the column headers are the same as in Figure 7-10. The serial running time T_S was 97.1 seconds. The throttling limit was $K = 10P$ for Cilk-P and $4P$ for Pthreads, because the Cilk-P implementation performed better with throttling limit of $10P$ than $4P$, whereas the Pthreaded implementation was the other way around.

Program	*Dependency Folding*	T_S	T_1	T_{16}	*Serial Overhead*	*Speedup* T_S/T_{16}	*Scalability* T_1/T_{16}
pipe-fib	no	20.7	23.5	4.7	1.13	4.40	4.98
pipe-fib-256	no	20.7	21.7	1.7	1.05	12.32	12.90
pipe-fib	yes	20.7	21.7	1.8	1.04	11.65	12.17
pipe-fib-256	yes	20.7	21.7	1.7	1.05	12.43	13.02

Figure 7-16: Performance evaluation using the *pipe-fib* benchmark on the AMD Opteron system. We tested the Cilk-P system with two different programs, the ordinary *pipe-fib*, and *pipe-fib-256*, which is coarsened. Each program is tested with and without the dynamic dependency folding optimization. For each program for a given setting, we show the running time of its serial counter part (T_S), running time executing on a single worker (T_1), on 16 workers (T_{16}), its serial overhead, scalability, and speedup obtained running on 16 workers.

number of processors. Unfortunately the answer depends on the architecture, since it is dictated by, for example, how much additional bookkeeping is necessary to enable parallel execution and how many cache misses are incurred due to data migration on a successful steal. Nevertheless, we believe that applications written in Cilk-P should generally scale as well as applications with comparable parallelism written using the baseline Cilk, since the additional work that the runtime performs for pipeline parallelism is within a small constant factor of the original scheduling overhead. Applications in Cilk-P also incur overhead in the form of cache misses to check cross edges. This overhead is akin to the spawn overhead in that one can amortize the overhead against the work done within the stage corresponding to the cross edge. Thus, as long as each stage contains substantial amount of work, this overhead should not impede scalability.

7.11 Pipeline throttling

What impact does throttling have on theoretical performance? PIPER relies on throttling to achieve its provable space bound and avoid runaway pipelines. Ideally, the user should not worry about throttling, and the system should perform well automatically, and indeed, PIPER's throttling of a pipeline computation is encapsulated in Cilk-P's runtime system. But what price is paid?

We can pose this question theoretically in terms of a pipeline computation G's ***unthrottled dag***: the dag $\widehat{G} = (V, \widehat{E})$ with the same vertices and edges as G, except without throttling edges. How does adding throttling edges to an unthrottled dag affect span and parallelism?

The following two theorems provide two partial answers to this question. We first consider ***uniform*** pipelines, which contain no hybrid stages and in which, for each stage j, node (i, j) is nearly identical across all iterations i. For uniform pipelines, we show that throttling does not affect the asymptotic performance of PIPER executing $\widehat{G}$.

Theorem 51 *Let $\widehat{G} = (V, \widehat{E})$ denote the unthrottled pipeline computation dag for a uniform linear pipeline program, and consider the execution of this program by* PIPER *on P processors with a throttling limit of $K = aP$ for some constant $a > 1$. Let $\widehat{T_1}$ and $\widehat{T_\infty}$ denote the work and span, respectively, of $\widehat{G}$. Then, for some sufficiently large constant x, the expected running time is $T_P \leq (1+c/a)\widehat{T_1}/P + c\widehat{T_\infty}$, and for any $\epsilon > 0$, with probability at least $1-\epsilon$, the running time is $T_P \leq (1 + c/a)\widehat{T_1}/P + c(\widehat{T_\infty} + \lg P + \lg(1/\epsilon))$.*

PROOF. For convenience, let us suppose that each stage has exactly the same cost in every iteration. It is straightforward to generalize the result such that the cost of a stage can vary by a constant amount between iterations.

We first establish some notation. Let n denote the number of iterations in $\widehat{G}$, and let m denote the number of stages in each iteration. Let W be the total work in each iteration. We assume that W includes the work performed within each stage in an iteration as well as the cost incurred to check any cross-edge dependencies from the previous iteration. Similarly, let S be the work of the most expensive serial stage in an iteration.

We can bound the work and span of $\widehat{G}$. The work of $\widehat{G}$ is $\widehat{T_1} = nW$. Meanwhile, any longest path from the beginning of any node $(i, 0)$ to the end of any node $(i + x, m - 1)$ has cost $W + xS$. In particular, any path connecting these nodes is represented by some interleaving of m vertical steps and x horizontal steps. For any path, the work of all vertical steps is exactly W, since the vertical steps must execute each stage 0 through $m-1$. Each of the x horizontal steps, meanwhile, executes exactly one serial stage, whose cost is at most S. Consequently, the length of any longest path from $(i, 0$ to $(i+x, m-1)$ is at most $W+xS$, and therefore the length of a longest path from $(0, 0)$ to $(n - 1, m - 1)$ is $\widehat{T_\infty} = W + nS$.

Now we bound the work T_1 and the span T_∞ of the throttled dag G in terms of the work and span of the unthrottled dag $\widehat{G}$. Because G adds $n - K$ zero-cost throttling edges to $\widehat{G}$, we have $T_1 = \widehat{T_1}$; the work remains the same. The throttling edges can increase the span, however. Consider a critical path ℓ through G, which has cost T_∞. Suppose this path ℓ contains $\kappa \geq 0$ throttling edges. Label the throttling edges along ℓ in order of increasing iteration number, with throttling edge k connecting the node terminal of (the subdag corresponding to the execution of node) $(i_k, m-1)$ to the node root of $(i_k+K, 0)$, for $1 \leq k \leq \kappa$. Removing all κ throttling edges from ℓ splits ℓ into $\kappa+1$ segments $\ell_0, \ell_1, \ldots, \ell_\kappa$. More precisely, let ℓ_0 denote the path from the beginning of node $(0, 0)$ to the end of $(i_1, m - 1)$, and let ℓ_k denote the path from the beginning of node $(i_k + K, 0)$ to the end of

$(i_{k+1}, m-1)$, where $i_{\kappa+1} = n-1$. By our previous result, the cost of ℓ_0 is $W + (i_1+1)S$, and the cost of each segment ℓ_k is $W + (i_{k+1} - i_k - K)S$. We thus have that T_∞, which is the total cost of ℓ, satisfies

$$\begin{aligned}
T_\infty &= W + (i_1+1)S + \sum_{k=1}^{\kappa} (W + (i_{k+1} - i_k - K)S) \\
&= W + \kappa W + (i_{\kappa+1}+1)S - \kappa K S \\
&= W + \kappa W + (n - \kappa K)S \\
&= W + nS + \kappa(W - KS) \\
&= \widehat{T_\infty} + \kappa(W - KS) \ .
\end{aligned}$$

A cut-and-paste argument shows that throttling edges in G can only increase the span if we have $W > KS$; otherwise, one can create an equivalent or longer path by going horizontally through K copies of the most expensive stage, rather than down an iteration i, and across a throttling edge that skips to the beginning of iteration $i + K$.

We now combine these bounds on T_1 and T_∞ with Theorem 49. We describe the analysis of the high-probability bound; the expectation bound follows similarly.

From Theorem 49, we know that, with probability $1 - \epsilon$, Piper executes G in time

$$\begin{aligned}
T_P &\le \frac{T_1}{P} + c\,(T_\infty + \lg P + \lg(1/\epsilon)) \\
&= \frac{\widehat{T_1}}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\epsilon)) + c\kappa(W - KS) \ .
\end{aligned}$$

If $\kappa = 0$, then the extra $c\kappa(W - KS)$ term is 0, giving the desired bound. Otherwise, assume $\kappa > 0$. Since every throttling edge skips ahead K iterations, we know that the critical path uses at most $\kappa < n/K$ throttling edges. Using this bound for κ and letting $K = aP$ for some constant $a > 1$, we can rewrite the expression for T_P as

$$\begin{aligned}
T_P &\le \frac{\widehat{T_1}}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\epsilon)) + \frac{cn}{aP}(W - aPS) \\
&= \frac{\widehat{T_1}}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\epsilon)) + \left(\frac{c}{a}\right)\left(\frac{nW}{P}\right)\left(1 - \frac{aPS}{W}\right) \\
&= \frac{\widehat{T_1}}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\epsilon)) + \left(\frac{c}{a}\right)\left(\frac{\widehat{T_1}}{P}\right)\left(1 - \frac{aPS}{W}\right) \\
&= \left(1 + \frac{c}{a}\left(1 - \frac{aPS}{W}\right)\right)\frac{\widehat{T_1}}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\epsilon)) \ .
\end{aligned}$$

The theorem follows from the fact that, for throttling edges to be included in a critical path, we must have $W \ge KS$, and therefore $W \ge aPS$. □

Second, we consider ***nonuniform*** pipelines, where the cost of a node (i,j) can vary across iterations. It turns out that nonuniform pipelines can pose performance problems, not only for Piper, but for any scheduler that uses a small amount of space. Figure 7-17 illustrates the dag for a pathological nonuniform pipeline for any scheduler that uses throttling. In this dag, T_1 work is distributed across $(T_1^{1/3} + T_1^{2/3})/2$ iterations such that any $T_1^{1/3} + 1$ consecutive iterations consist of 1 ***heavy*** iteration, with $T_1^{2/3}$ work, and $T_1^{1/3}$

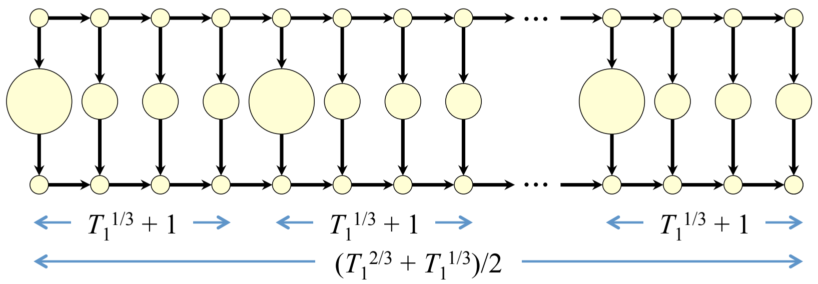


Figure 7-17: Sketch of the pathological unthrottled linear pipeline dag, which can be used to prove Theorem 52. Small circles represent nodes with unit work, medium circles represent nodes with $T_1^{1/3} - 2$ work, and large circles represent nodes with $T_1^{2/3} - 2$ work. The number of iterations per cluster is $T_1^{1/3} + 1$, and the total number of iterations is $(T_1^{2/3} + T_1^{1/3})/2$.

light iterations of $T_1^{1/3}$ work each. Intuitively, achieving a speedup of 3 on this dag requires having at least 1 heavy iteration and $\Theta(T_1^{1/3})$ light iterations active simultaneously, which is impossible for any scheduler that uses a throttling limit of $K = o(T_1^{1/3})$. The following theorem formalizes this intuition.

Theorem 52 *Let $\widehat{G} = (V, \widehat{E})$ denote the nonuniform unthrottled linear pipeline computation dag shown in Figure 7-17, with work T_1 and span $T_\infty \leq 2T_1^{2/3}$. Let S_1 denote the optimal stack-space usage when $\widehat{G}$ is executed on 1 processor. Any P-processor execution of $\widehat{G}$ that achieves $T_P \leq T_1/\rho$, where ρ satisfies $3 \leq \rho \leq O(T_1/T_\infty)$, uses space $S_P \geq S_1 + (\rho - 2)T_1^{1/3}/2$.*

PROOF. Consider the pipeline computation shown in Figure 7-17. Suppose that a scheduler executing the pipeline dag requires $S_1 + x - 1$ space to execute x iterations of the pipeline simultaneously, that is, $S_1 - 1$ stack space to execute the function containing the pipeline, plus unit space per pipeline iteration the scheduler executes in parallel. Consequently, the scheduler executes the pipeline serially using S_1 space, and incurs an additional unit space overhead per pipeline iteration it executes in parallel. Furthermore, suppose that each node is a serial computation, that is, no nested parallelism exists in the nodes of Figure 7-17.

Consider a time step during which the scheduler is executing instructions from k heavy iterations in parallel. Because stage 0 of the pipeline in Figure 7-17 is serial, to execute instructions from k heavy iterations in parallel requires executing the node for stage 0 in at least $(k-1)T_1^{1/3} + 1$ consecutive iterations. Because stage 2 is serial, the scheduler must have executed stage 0, but not stage 2, for at least $(k-1)T_1^{1/3} + 1$ iterations. Consequently, the scheduler requires at least $S_P \geq S_1 + (k-1)T_1^{1/3}$ stack space to execute instructions from k heavy iterations in parallel.

We now bound the number k of heavy iterations the scheduler must execute in parallel to achieve a speedup of ρ. The total time T_P the scheduler takes to execute the instructions of the pipeline in Figure 7-17 is at least the total time it takes to execute all $T_1^{2/3} \cdot T_1^{1/3}/2 = T_1/2$ instructions in heavy iterations. Assuming the scheduler executes instructions from k heavy iterations simultaneously on each time step it executes any instruction from a heavy iteration, we have $T_P \geq T_1/(2k)$. Rearranging terms gives us that $k \geq T_1/(2T_P) = \rho/2$.

Combining these bounds shows that the scheduler requires at least $S_P \geq S_1 + (\rho - 2)T_1^{1/3}/2$ space to achieve a speedup of ρ when executing the pipeline in Figure 7-17. □

Intuitively, these two theorems present two extremes of the effect of throttling on pipeline dags. One interesting avenue for research is to determine what are the minimum restrictions on the structure of an unthrottled linear pipeline G that would allow a scheduler to achieve parallel speedup on P processors using a throttling limit of only $\Theta(P)$.

7.12 Conclusion

This chapter introduces Cilk-P, a system that extends the Cilk parallel-programming model to support on-the-fly pipeline parallelism. In this chapter, we describe new language constructs — the `pipe_while` loop and the `pipe_stage_wait` and `pipe_stage` statements — that extend fork-join parallel languages such as Cilk to support on-the-fly pipeline parallelism. Cilk-P thus supports principled approaches to writing programs that exhibit pipeline parallelism. We present PIPER, a work-stealing scheduler that executes computations containing `pipe_while` loops in a provably time- and space-efficient manner. We have incorporated these linguistics and the scheduler into a prototype Cilk-P implementation. We demonstrate effectiveness of Cilk-P in practice by parallelizing three benchmarks from the PARSEC suite using our Cilk-P prototype and showing that these implementations are competitive with alternative versions coded using TBB or Pthreads. In particular, we show that `pipe_while` loops are expressive enough for parallelizing *x264*, a benchmark with a complicated pipelining structure which is difficult to express using the pipeline model supported by TBB. By employing an efficient implementation of PIPER, our Cilk-P prototype thus supports a theory of performance for pipeline programs that is borne out in practice. Cilk-P thus support a scientific approach to writing and reasoning about programs that exhibit pipeline parallelism.

Our investigation also highlights several limitations of our language constructs, which represent potential areas for future study. First, because our constructs only allow programmers to specify dependencies between consecutive `pipe_while` loop iterations, it is natural to ask whether one might extend the loop construct to support more generic pipelines. For example, one could imagine extending the `pipe_stage_wait` construct to allow for dependencies on the same stage in multiple preceding iterations, or perhaps even different stages from preceding iterations. In our study, we chose to limit dependencies to only adjacent iterations because this limitation simplifies the semantics of the language construct for pipelining and allows for an efficient implementation. The design of PIPER and the lazy enabling optimization both depend on the fact that dependencies are only between consecutive iterations. In a `pipe_while` loop, when computation of a node in iteration i finishes, the only node in a future iteration that might be enabled is node in iteration $i + 1$. Both the runtime implementation and the mathematical analysis become more complicated if nodes in multiple iterations might be enabled by the completion of node in iteration i. On the other hand, extending the loop construct to allow dependencies within a constant-sized sliding window of iterations might be useful for some applications. An interesting open question is whether one can support a more expressive pipeline loop construct without a significant increase in complexity in the linguistic interface or runtime implementation.

It might also be interesting to study how `pipe_while` loops might simplify programs written using other parallel programming models or runtimes. In this work, we focus on integrating `pipe_while` loops with Cilk-like languages, which typically focus on fork-join parallelism and utilize a work-stealing scheduler. The `pipe_while` construct itself is agnostic, however, to the scheduling technique used by the underlying runtime. For example, the semantics of a `pipe_while` loop should be equally useful for programs written using OpenMP, which more often than not assume a work-sharing environment. Depending on the nature of the workload and the available parallelism in the application, alternative scheduler implementations might improve performance.

Finally, an open question about pipeline parallelism is whether one might be able to improve the interaction of `pipe_while` loops with file I/O. For instance, the Pthreaded implementation of *dedup* seems to gain a performance advantage by using a single designated thread to handle the file I/O. In a dynamic multithreaded language such as Cilk-P, the file I/O stage for different iterations likely ends up being processed by different workers, because the runtime does not distinguish between I/O stages and computation stages when scheduling a `pipe_while` loop. In principle, one might see improved performance if the scheduler and runtime are aware of the I/O characteristics of pipelines and schedule accordingly.

7.13 Recent developments

Several developments related to this work have occurred since its publication. Intel used this work on Cilk-P to produce an open-source prototype library that supports "on-the-fly" pipeline parallelism [379, 380]. Dimitrov *et al.* developed a provably good determinacy-race detector for programs whose computations can be modeled as a 2D lattice, which includes Cilk-P programs. With regards to handling expensive I/O operations efficiently, as was an issue for the *dedup* benchmark, Muller and Acar presented a randomized work-stealing scheduler that effectively hides the latency of such operations [291].

Chapter 8

The Cilkprof Scalability Profiler

This chapter presents the Cilkprof scalability profiler [346]. This work was conducted in collaboration with Bradley C. Kuszmaul, I-Ting Angelina Lee, Charles E. Leiserson, and William M. Leiserson.

8.1 Introduction

When a Cilk multithreaded program fails to attain linear speedup when scaling up to large numbers of processors, there are four common reasons [180]:

***Insufficient parallelism*:** The program contains serial bottlenecks that inhibit its scalability.

***Scheduling overhead*:** The work that can be done in parallel is too fine grained to be worth distributing to other processors.

***Insufficient memory bandwidth*:** The processors simultaneously access memory (or a level of cache) at too great a rate for the bandwidth of the machine's memory network to sustain.

***Contention*:** A processor is slowed down by simultaneous interfering accesses to synchronization primitives, such as mutex locks, or by the true or false sharing of cache lines.

Performance engineers can benefit from profiling tools that identify where in their program code these problems might be at issue, as well as eliminate consideration of code that does not have issues so that the detective work can be properly focused elsewhere. Profiling tools measure each part of a program to give performance engineers visibility into where the program spends its execution time or computing resources. These tools thus enable a principled approach to finding performance bottlenecks and directing optimization effort. This chapter introduces a scalability profiler, called Cilkprof, which can help identify the causes of insufficient parallelism and scheduling overhead in a Cilk multithreaded program.

Cilkprof builds on the approach taken by Cilkview [180], which measures the work and span of a Cilk computation and reports on its parallelism. To help programmers diagnose scalability bottlenecks, Cilkview provides an API to control which portions of a Cilk program should be analyzed. This API allows a programmer to restrict Cilkview's analysis by designating "start" and "stop" points in the code, similarly to the how the programmer can measure the execution time of various portions of a C program by inserting `gettimeofday` calls. But using this API requires the programmer to manually probe portions of the code, which can be cumbersome and error prone for large and complex codebases, such as codebases that contain recursive functions.

In contrast, Cilkprof profiles the parallelism, much as gprof [171] profiles execution time. Unlike gprof, however, which uses asynchronous sampling, and Cilkview, which uses dynamic binary instrumentation using Pin [267], Cilkprof uses compiler instrumentation (see, for example, [350,351]) to gather detailed information about a Cilk computation. Conceptually, during a serial run of an instrumented Cilk program, Cilkprof analyzes every ***call site***, that is, every location in the code where a function is either called or spawned. It determines how much of the work and span of the overall computation is attributable to the subcomputation that begins when the function invoked at that call site is called or spawned and that ends when that function returns. Cilkprof calculates work and span in terms of processor cycles, but it can also use other measures such as execution time, instruction count, cache misses, etc. Cilkprof's analysis allows a programmer to evaluate the scalability of that call site — the scalability of the computation attributable to that call site — and how it affects the overall computation's scalability. In particular, by embedding work-span analysis into its evaluation of program scalability, Cilkprof produces a profile whose results conform to the theory of performance that programmers use to reason about the scalability of Cilk programs.

Although we implemented Cilkprof to analyze Cilk Plus [196] programs, in principle, the same tool could be implemented for any of the variants of Cilk, including MIT Cilk [146] or Cilk++ [246]. More generally, the Cilkprof algorithm could be adapted to profile any parallel program whose span can be computed during a serial execution. Because Cilkprof runs on a serial execution of the program under test, it does not capture variations in work and span that may occur in a nondeterministic program.

Cilkprof can help Cilk programmers quickly identify scalability bottlenecks within their programs. We used Cilkprof to analyze our implementation of PBFS, an 1800-line code that is described in Chapter 3. After about two hours of poring over Cilkprof data, we were able to identify a serial bottleneck within PBFS, fix it, and confirm that our modification improved parallelism by a factor of 5. Cilkprof allowed us to eliminate insufficient parallelism as the code's scalability bottleneck and, thereby, to focus on the real bottleneck, which is memory bandwidth. Section 8.8 describes this case study.

Efficiently computing the work and span of every call site is harder than it may appear. Let us consider some simple approaches.

Suppose first that we measure the execution time of every strand and aggregate the results after the execution completes. Although smaller than the total number of executed instructions, these data would be huge for many parallel applications, with space rivaling T_1, the normal serial running time of the program being analyzed. The data thus cannot reasonably be stored for later analysis, and the computation must be performed on the fly.

Intuitively, however, computing these results on the fly poses its own challenges. Because a strand's execution affects all the call sites on the call stack, a naive strategy could potentially blow up the running time to as much as $\Theta(DT_1)$, where D is the maximum depth of the call stack. Of course, if a function `f` calls another function `g`, then the profile for `f` must include the profile for `g`. We could therefore compute local profiles for each function and update the parent with the profile of the child whenever a child returns, but this strategy could be just as bad or worse than updating each function on the call stack. If the profile contains S call sites, each function return could involve $\Theta(S)$ work, blowing up the running time to as much as $\Theta(ST_1)$. Furthermore, even if one computed the work with these methods, computing the span, which is similar to computing the longest path in a directed acyclic graph, would add considerable complexity to the computation.

By using a carefully constructed algorithm and an amortized `prof` data structure to represent profiles, Cilkprof computes work and span profiles with remarkable alacrity. Theoretically, Cilkprof computes the profiles in $\Theta(T_1)$ time and $O(DS)$ space, where T_1 is the work of the original Cilk program, D is the maximum call-stack depth, and S is the number of call sites in the program. In practice, the overheads are strikingly small. We implemented Cilkprof by instrumenting the Cilk Plus/LLVM compiler [195], a branch of the LLVM compiler that contains the Cilk linguistic extensions. On a set of 16 application benchmarks, Cilkprof incurred a geometric-mean multiplicative slowdown of 1.9 and a maximum slowdown of 7.4, compared to the uninstrumented serial running times of these benchmarks.

Cilkprof's measurements seem ample for debugging scalability bottlenecks. Naturally, the Cilkprof instrumentation introduces some error into measurements of the program under test. Cilkprof compensates by subtracting estimates of its own overhead from the work and span measurements it gathers in order to reduce the effects of compiler instrumentation. Furthermore, it generally suffices to measure the parallelism of a program to within a binary order of magnitude in order to diagnose whether the program suffers from insufficient parallelism [180]. The overhead introduced by Cilkprof instrumentation appears to deliver work and span numbers within this range. Moreover, if one considers the errors in work and span to be similarly biased, the computation of their ratio, the parallelism, should largely cancel them.

Cilkprof as described herein does not have a sophisticated user interface. The current Cilkprof "engine" simply dumps the computed profile to a file in comma-separated-value format suitable for inputting to a spreadsheet. We view the development of a compelling user interface for Cilkprof as an open research question.

This chapter makes the following contributions:

- The Cilkprof algorithm for computing the work and span attributable to each call site in a program, which provably operates with only constant overhead.
- The `prof` data structure for supporting amortized $\Theta(1)$-time updates to profiles.
- An implementation of Cilkprof which runs with little slowdown compared to the uninstrumented program under test.
- Two case studies — parallel quicksort and parallel breadth-first search of a graph — demonstrating how Cilkprof can be used to diagnose scalability bottlenecks.

This remainder of this chapter is organized as follows. Section 8.2 illustrates how the profile data computed by Cilkprof can help to analyze the scalability of a simple parallel quicksort program. Sections 8.3 and 8.4 describe how the Cilkprof algorithm works and proves that Cilkprof incurs $\Theta(1)$ amortized overhead per program instruction. Section 8.5 presents an implementation of the `prof` data structure and shows that profile statistics can be updated in $\Theta(1)$ amortized time. Section 8.6 describes the profile of work and span measurements that Cilkprof computes for a Cilk program. Section 8.7 overviews the implementation of Cilkprof and analyzes its empirical performance. Section 8.8 describes how Cilkprof was used to diagnose a scalability bottleneck in PBFS. Section 8.9 discusses Cilkprof's relationship to related work, and Section 8.10 provides some concluding remarks.

8.2 Parallel quicksort

This section illustrates the usefulness of Cilkprof by means of a case study of a simple parallel quicksort program coded in Cilk. Although the behavior of parallel quicksort is well understood theoretically, the profile data computed by Cilkprof allows a programmer

```
01 int partition(long array[], int low, int high) {
02   long pivot = array[low + rand(high - low)];
03   int l = low - 1;
04   int r = high;
05   while (true) {
06     do { ++l; } while (array[l] < pivot);
07     do { --r; } while (array[r] > pivot);
08     if (l < r) {
09       long tmp = array[l];
10       array[l] = array[r];
11       array[r] = tmp;
12     } else {
13       return (l == low ? l + 1 : l);
14     }
15   }
16 }
17
18 void pqsort(long array[], int low, int high) {
19   if (high - low < COARSENING) {
20     // base case: sort using insertion sort
21   } else {
22     int part = partition(array, low, high);
23     cilk_spawn pqsort(array, low, part);
24     pqsort(array, part, high);
25     cilk_sync;
26   }
27 }
28
29 int main(int argc, char *argv[]) {
30   int n;
31   long *A;
32   // parse arguments
33   // initialize array A of size n
34   pqsort(A, 0, n);
35   // do something with A
36   return 0;
37 }
```

Figure 8-1: Cilk code for a parallel quicksort that sorts an array of 64-bit integers. The variable `COARSENING` is a constant defining the maximum number of integers to sort in the base case. We used `COARSENING=32`.

to diagnose quicksort's partitioning subroutine as serial bottleneck without understanding the theoretical analysis.

Figure 8-1 shows the Cilk code for a quicksort [186] program that has been parallelized using the Cilk parallel keywords `cilk_spawn` and `cilk_sync`. The `cilk_spawn` on line 23 spawns the recursive call to `pqsort`, allowing this `pqsort` instantiation to execute in parallel with the recursive call to `pqsort` on line 24. In principle, the call to `pqsort` on line 24 could also have been spawned, but since the continuation of that call does nothing but synchronize the children, spawning the call would not increase the parallelism and would increase the overhead. The `cilk_sync` on line 25 ensures that the computation performed by the spawn on line 23 finishes before `pqsort` returns.[1]

Parallel quicksort provides a good example to illustrate what Cilkprof does, because its behavior is well understood theoretically. With high probability, `pqsort` performs $\Theta(n \log n)$ work to sort an array of n elements. The call to `partition` in line 22 performs $\Theta(n)$ work

[1] Although the `cilk_sync` on line 25 is not strictly necessary, because functions in Cilk automatically sync when they return, we include the `cilk_sync` in this code to clarify when `pqsort` syncs.

Line	T_1	T_∞	T_1/T_∞
22	408,150,528	408,150,528	1.0
23	741,312,781	116,591,841	6.4
24	761,041,165	125,360,000	6.1
34	790,518,060	141,902,681	5.6

Figure 8-2: A subset of the on-work profile that Cilkprof reports for running the parallel quicksort code in Figure 8-1 to sort an array of 10 million random 64-bit integers. The on-work profile records data for all instantiations in the computation. For each call site, the "T_1" column gives the sum of the work of all invocations of that call site, and the "T_∞" column gives the sum of the spans of those invocations. The "T_1/T_∞" column gives the parallelism of each call site, as computed from the "T_1" and "T_∞" values for that call site. All times are measured in nanoseconds.

Line	T_1	T_∞	T_1/T_∞	*Local* T_1	*Local* T_∞
22	141,891,291	141,891,291	1.0	141,891,291	141,891,291
23	597,298,216	98,119,730	6.1	4,340	3,823
24	691,808,220	118,447,199	5.8	7,068	6,682
34	790,518,060	141,902,681	5.6	885	885

Figure 8-3: A subset of the on-span profile that Cilkprof reports for running the parallel quicksort code in Figure 8-1 to sort an array of 10 million random 64-bit integers. The on-span profile records data only for instantiations that fall on the critical path of the computation. For each call site, the "T_1," "T_∞" and "T_1/T_∞" columns are similar to their on-work counterparts, shown in Figure 8-2. The "Local T_1" column contains, for each call site, the cumulative work of all invocations of that call site on the critical path, excluding all work in children of the instantiated function. The "Local T_∞" column is similar, except that it presents the cumulative span. All times are measured in nanoseconds.

to partition an array of n elements and is a major contributor to the critical path of the computation, precluding `pqsort` from exhibiting more than $O(\log n)$ parallelism. (For a similar analysis of merge sort, see [100, Ch. 27.3].) A more careful analysis — one that pays attention to the constants hidden inside the big-Oh — indicates that on an array of 10 million elements, `pqsort` exhibits a parallelism of approximately $\ln 10^6 = 16$. To achieve linear speedup, however, a program should exhibit substantially more parallelism than there are processors on the machine [146]. This parallel quicksort program has too little parallelism to keep more than a few processors busy.

Parallel quicksort's scalability profile

Suppose that we did not already know where the serial bottleneck in the code in Figure 8-1 lies, however. Let us see how we can use Cilkprof to discover that `partition` is the main culprit.

Figures 8-2 and 8-3 present an excerpt of the data Cilkprof reports from running `pqsort` on an array of 10 million 64-bit integers, cleaned up for didactic clarity. Cilkprof computes two "profiles" for the computation: an "on-work profile" and an "on-span profile." Each ***profile*** contains a record of work and span data for each call site in the computation. A record in the ***on-work*** profile (Figure 8-2) accumulates work and span data for every invocation of a particular call site in the computation. A record in the ***on-span*** profile (Figure 8-3) accumulates work and span data only for the invocations of a particular call site that appear on the critical path of the computation. Section 8.6 describes precisely what work and span values each record stores and how Cilkprof accommodates recursive functions.

Let us explore the data in Figures 8-2 and 8-3 to see what these data tell us about the scalability of this quicksort code. The on-work profile shows us that the work and span of the computation is dominated by line 34, the instantiation of `pqsort` from `main`. The "T_1/T_∞" value in Figure 8-2 for this line tells us that this call to `pqsort` exhibits a parallelism of only 5.6, even less than the 16-fold parallelism that our analysis predicted. To see why this call to `pqsort` exhibits poor parallelism, we can examine what different call sites contribute to the span of the computation.

Let us start by examining Cilkprof's "local T_∞" data in its on-span profile, Figure 8-3. Conceptually, the "local T_∞" for a call site s that calls or spawns a function `f` specifies how much of the span comes from instructions executed under s, not including instructions executed under `f`'s call sites. For the quicksort code in Figure 8-1, we can observe two properties of these "local T_∞" data. First, the sum of the "local T_∞" values in the on-span profile (Figure 8-3) for the three call sites in `pqsort` (lines 22, 23, and 24) and the call to `pqsort` from `main` (line 34) equals the "T_∞" value in the on-work profile (Figure 8-2) for the call to `pqsort` from `main`. These four call sites therefore account for the entire span of line 34. Second, line 22 in the on-span profile, the "local T_∞" of accounts for practically all of the span of line 34, indicating that line 22 is the parallelism bottleneck for the instantiation of `pqsort` from `main`.

What else does Cilkprof tell us about line 22? The "T_1/T_∞" for line 22 in the on-span profile shows that all instances of this call site on the critical path are serial. Consequently, parallelizing this call site is key to improving the parallelism of the computation. From examining the code, we therefore conclude that we must parallelize `partition` to improve the scalability of `pqsort`, as we expect from our understanding of quicksort's theoretical performance. Cilkprof's data allows the serial bottleneck in quicksort to be identified without prior knowledge of its analysis.

8.3 Computing work and span

This section describes how Cilkprof computes the work and span of a Cilk computation. Cilkprof's algorithm for work and span is based on a similar algorithm from [180]. After defining some useful concepts, we describe the "work-span" variables used to perform the computation. We give the algorithm and describe the invariants it maintains. We show that on a Cilk program under test that executes in T_1 work and has stack depth D, Cilkprof's work-span algorithm runs in $O(T_1)$ time using $O(D)$ extra storage. Section 8.4 will extend this work-span algorithm to compute profiles.

Definitions

Let us first define some terms. The program under test is a Cilk binary executable containing a set I of ***instructions***. Some of the instructions in I are ***functions*** — they can be called or spawned — and some are ***call sites*** — they call or spawn a function. The (mathematical) function φ maps a call site to the function in which the call site resides.

When the program is executed serially, it produces a sequence XI of ***executed instructions***. The function $\sigma : XI \to I$ indicates which instruction $i \in I$ was executed to produce a given executed instruction $xi \in XI$. A contiguous subsequence of instructions in XI is called a ***trace***. For a given executed call site $xi \in XI$, the ***trace*** of xi, denoted $\text{Trace}(xi)$, is the contiguous subsequence of XI starting with xi's successor — the first instruction of the

F spawns or calls G:

```
1  $G.w = 0$
2  $G.p = 0$
3  $G.\ell = 0$
4  $G.c = 0$
```

Called G returns to F:

```
5  $G.p \mathrel{+}= G.c$
6  $F.w \mathrel{+}= G.w$
7  $F.c \mathrel{+}= G.p$
```

Spawned G returns to F:

```
8   $G.p \mathrel{+}= G.c$
9   $F.w \mathrel{+}= G.w$
10  if $F.c + G.p > F.\ell$
11      $F.\ell = G.p$
12      $F.p \mathrel{+}= F.c$
13      $F.c = 0$
```

F syncs:

```
14  if $F.c > F.\ell$
15      $F.p \mathrel{+}= F.c$
16  else
17      $F.p \mathrel{+}= F.\ell$
18  $F.c = 0$
19  $F.\ell = 0$
```

F executes an instruction:

```
20  $F.w \mathrel{+}= 1$
21  $F.c \mathrel{+}= 1$
```

Figure 8-4: Pseudocode for Cilkprof's work-span algorithm. For simplicity, this pseudocode computes work and span by incrementing the work and continuation at each instruction, rather than by any of several more efficient methods to compute instruction counts.

executed function that was called or spawned — and ending with the corresponding return from the executed function.

For simplicity, assume that work and span are measured by counting instructions. It is straightforward to adapt the Cilkprof algorithm to measure work and span in terms of processor cycles, execution time, or even cache misses and other measures. The ***work*** of a trace T, denoted $\text{Work}(T)$, is the number of instructions in T. The ***span*** of a trace T, denoted $\text{Span}(T)$, is the maximum number of instructions along any path of dependencies from the first instruction in T to the last instruction in T.

Work-span variables

Cilkprof measures the work and span of a Cilk computation in a manner similar to the Cilkview algorithm [180]. As Cilkprof serially executes the Cilk program under test, it computes the work and span of each instantiated function.

For each instantiated function F, four ***work-span variables*** are maintained in a frame for F on a ***shadow stack*** which is pushed and popped in synchrony with the function-call stack. Let T denote the trace of F executed so far. The ***work*** variable $F.w$ corresponds to the work on T. The remaining three ***span variables*** are used to compute the span of F. Conceptually, Cilkprof maintains the executed `cilk_spawn` instruction u in T that, since F last synced, spawned the child instantiation of F that realizes the span of T. The location u is not explicitly maintained, however, but the values of the three variables reflect its position in T. Specifically, the three span variables are defined as follows:

- The ***prefix*** $F.p$ stores the span of the trace starting from the first instruction of F and ending with u. The path that realizes $F.p$ is guaranteed to be on the critical path of F.
- The ***longest-child*** $F.\ell$ stores the span of the trace from the start of F through the return of the child that F spawns at u.
- The ***continuation*** $F.c$ stores the span of the trace from the continuation of u through the most recently executed instruction in F.

The work-span algorithm

Figure 8-4 gives the pseudocode for the basic Cilkprof algorithm for computing work and span. At any given moment during Cilkprof's serial execution of the program under test, each nonzero work-span variable z holds a value corresponding to a trace, which we define as the ***trace*** of the value and denote by $\text{Trace}(z)$. The pseudocode maintains three invariants:

Invariant 53 *The trace of the value in a variable is well defined, that is, it is a contiguous subsequence of XI.*

Invariant 54 *If z is a work variable, then $z = \text{Work}(\text{Trace}(z))$.*

Invariant 55 *If z is a span variable, then $z = \text{Span}(\text{Trace}(z))$.*

These invariants can be verified by induction on instruction count by inspecting the pseudocode in Figure 8-4. For example, just before G returns from a spawn, we can assume inductively that $G.p$ and $G.c$ hold the spans of their traces. At this point, the trace of $G.p$ starts at the first instruction of G and ends with u, traversing all called and spawned children in between. The trace of $G.c$ starts at the continuation of u and continues to the current instruction, also traversing all called and spawned children in between. Thus, they have explored the entire trace of G between them. Consequently, when line 8 executes, the trace of $G.p$ becomes the entire trace of G, and $G.p$ becomes $\text{Span}(\text{Trace}(G.p))$, maintaining the invariants. Other code sequences succumb to similar reasoning.

Performance

The next theorem bounds the running time and space usage of Cilkprof's algorithm for computing work and span.

Lemma 56 *Cilkprof computes the work and span of a Cilk computation in $\Theta(T_1)$ time using $\Theta(D)$ space, where T_1 is the work of the Cilk computation and D is the maximum call-stack depth of the computation.*

Proof. Inspection of the pseudocode from Figure 8-4 reveals that a constant number of operations on work-span variables occur at each function call or spawn, each sync, and each function return in the computation. Consequently, the running time is $\Theta(T_1)$. Since there are 4 work-span variables in each frame of the shadow stack, the space is $\Theta(D)$. □

8.4 The basic profile algorithm

This section describes the basic algorithm that Cilkprof uses to compute profiles for a Cilk computation. We first introduce the abstract interface for the `prof` data structure, whose implementation is detailed in Section 8.5. We show how to augment the work-span algorithm to additionally compute profiles for the computation. We analyze Cilkprof under the assumption, borne out in Section 8.5, that the `prof` data structure supports all of its methods in $\Theta(1)$ amortized time. We show that Cilkprof executes a Cilk computation in $O(T_1)$ time, where T_1 is the work of the original Cilk computation.

The prof *data structure*

As it computes the work and span of a Cilk computation, Cilkprof updates profiles in a `prof` data structure, which records work and span data for each call site in the computation. Let us see how Cilkprof computes these profiles, in terms of the abstract interface to the `prof` data structure. Section 8.5 describes how a `prof` can be implemented efficiently.

The `prof` data structure is a key-value store R that maintains a set of ***key-value pairs*** $\langle s, v\rangle$ as elements, where the key s is a call site and the value v is a record containing a ***work field*** $v.\mathit{work}$ and a ***span field*** $v.\mathit{span}$. The following methods operate on `prof`'s:

- Init(R): Initialize `prof` R to be an empty profile, deleting any key-value pairs stored in R.
- Update($R, \langle s, v\rangle$): If no element $\langle s, v'\rangle$ already exists in R, store $\langle s, v\rangle$ into R. If such an element exists, store $\langle s, v' + v\rangle$, where corresponding fields of v' and v are summed.
- Assign(R, R'): Move the contents of `prof` R' into R, deleting any old values in R, and then initialize R'.
- Union(R, R'): Update the `prof` R with all the elements in the `prof` R', and then initialize R'.
- Print(R): List all the key-value pairs in the `prof` R, and initialize R.

We shall show in Section 8.5 that each of these methods can be implemented to execute in $\Theta(1)$ amortized time.

Profiles

What profiling data does Cilkprof compute? Consider a call site s. During a serial execution of the program, the function $\varphi(s)$ containing s may call or spawn a function (or functions, if the target of the call or spawn is a function pointer) at s. Let OW be the set of executed call sites for which $xi \in OW$ implies that $\sigma(xi) = s$. For each $xi \in OW$, recall that Trace(xi) is the set of instructions executed after the call or spawn at the call site until the corresponding return. Intuitively, the work-on-work for s is the total work of all of these calls, which is to say

$$\sum_{xi \in OW} \text{Work}(\text{Trace}(xi)) \ ,$$

and the span-on-work for s is

$$\sum_{xi \in OW} \text{Span}(\text{Trace}(xi)) \ .$$

The work-on-span and span-on-span for s are similar, where the sum is taken over OS, the set of instructions along the span of the computation for which $xi \in OS$ implies that $\varphi(xi) = s$. These definitions are inadequate, however, for recursive codes, because two instantiations of s on the call stack cause double counting. Rather than complicate the explanation of the algorithm at this point, let us defer the issue of recursion until Section 8.6 and assume for the remainder of this section that no recursive calls occur in the execution, in which case these profile values for s are accurate.

Cilkprof computes these profile values for all call sites in the program by associating a `prof` data structure with each work-span variable in Figure 8-4. For a variable z, let $z.\mathit{prof}$ denote z's `prof`, let $z.\mathit{prof}[s].\mathit{work}$ denote the value of the *work* field for a call site $s \in I$ in the profile data for z, and let $z.\mathit{prof}[s].\mathit{span}$ denote the value of $z.\mathit{prof}$'s *span* field for s.

The Cilkprof algorithm

As Cilkprof performs the algorithm in Figure 8-4, in addition to computing the work and span, it also updates the prof associated with each work-span variable. First, just before each of lines 6 and 9, Cilkprof executes

$$\textsc{Update}(G.w.\mathit{prof}, \langle s, [\mathit{work}\colon G.w,\ \mathit{span}\colon G.p]\rangle)$$
$$\textsc{Update}(G.p.\mathit{prof}, \langle s, [\mathit{work}\colon G.w,\ \mathit{span}\colon G.p]\rangle)\ ,$$

where s is the call site where F spawns or calls G. In addition, Cilkprof performs the following calculations, where y and z denote two distinct variables in the pseudocode:

- Whenever the pseudocode assigns $y = 0$, Cilkprof also initializes its prof, executing $\textsc{Init}(y.\mathit{prof})$.
- Whenever the pseudocode assigns $y = z$, Cilkprof also assigns z's prof to y's prof, executing $\textsc{Assign}(y.\mathit{prof}, z.\mathit{prof})$.
- Whenever the pseudocode executes $y \mathrel{+}= z$, Cilkprof also unions their prof's, executing $\textsc{Union}(y.\mathit{prof}, z.\mathit{prof})$.

As a point of clarification, executing lines 20–21 causes no additional calculations to be performed on prof data structures, because 1 is not a variable.

Correctness

Recall that each work-span variable z in Figure 8-4 defines a trace Trace(z). For each variable z and call site $s \in I$, Cilkprof maintains the invariant

$$z.\mathit{prof}[s].\mathit{work} = \sum_{xi \in \text{Trace}(z)\ :\ \sigma(xi)=s} \text{Work}(\text{Trace}(xi))$$

and a similar invariant for $z.\mathit{prof}[\text{s}].\mathit{span}$. One can verify by induction on instruction count that the Cilkprof algorithm maintains these invariants.

Analysis of performance

The next theorem bounds the running time and space usage of Cilkprof. This analysis assumes that all prof methods execute in $\Theta(1)$ amortized time and that a single prof occupies $O(S)$ space, where S is the number of call sites in the Cilk computation. Section 8.5 shows how prof can achieve these bounds.

Theorem 57 *Cilkprof executes a Cilk computation in $\Theta(T_1)$ time using $O(DS)$ space, where T_1 is the work of the Cilk computation, D is the maximum stack depth of the computation, and S is the number of call sites in the computation.*

Proof. By Lemma 56, the work-span algorithm contributes negligibly to either time or space, and so it suffice to analyze the contributions due to method calls on the prof data structure. Inspection of the pseudocode from Figure 8-4, together with the modifications to make it handle profiles, reveals that a constant number of operations on work-span variables occur at each function call or spawn, each sync, and each function return in the computation. Returning from a function causes a constant number of method calls on the prof to be performed, and each operation on a variable induces at most a constant number of method calls on its associated prof, each of which takes $\Theta(1)$ amortized time, as Theorem 58 in

Section 8.5 will show. Consequently, each operation performed by Cilkprof to compute the work and span incurs at most constant overhead, yielding $\Theta(T_1)$ for the total running time of Cilkprof.

The space bound is the product of the maximum depth D of function nesting and the maximum size of a frame on the shadow stack. Each frame of the shadow stack contains 4 work-span variables and their associated `prof` data structures, each of which has size at most S. Thus, since the size of a frame on the shadow stack is $O(S)$, the total space is $O(DS)$. $\square$

8.5 The prof data structure

This section describes how the the `prof` data structure employed by Cilkprof is implemented. We first assume that the number of call sites is known *a priori*. We investigate the problems that arise when implementing a `prof` as an array or linked list, and then we see how a hybrid implementation can achieve $\Theta(1)$ amortized time for all its methods. We then remove the assumption and extend `prof` to the situation when call sites are discovered dynamically on the fly while still maintaining a $\Theta(1)$ amortized time[2] for each of its methods.

The basic data structure

To simplify the description of the implementation of the `prof` data structure, assume for the moment that Cilkprof magically knows *a priori* the number S of call sites in the computation. The compiler sets up a global hash table h mapping each call site s to a distinct index $h(s) \in \{0, 1, \ldots, S-1\}$.

The `prof` data structure is a hybrid of two straightforward implementations: an array and a list. Separately, each implementation would use too much time or space, but in combination they yield the desired space and time.

The ***array implementation*** represents a `prof` R as an S-entry array $R.\mathit{arr}[0\,..\,S-1]$. In this implementation, the INIT method allocates a new S-entry array $R.\mathit{arr}$ and zeroes it, costing $\Theta(S)$ time. The call $\textsc{Update}(R, \langle s, v\rangle)$ updates the entry with $R.\mathit{arr}[h(s)]+v$ (where $+$ performs fieldwise addition on the *work* and *span* fields of the records), taking only $\Theta(1)$ time. $\textsc{Union}(R, R')$ iterates through the entries of R' and updates the corresponding entries in R, zeroing R' as it goes, costing $\Theta(S)$ time. Finally, $\textsc{Assign}(R, R')$ iterates through the arrays copying the elements of R' to R, zeroing R' as it goes, also costing $\Theta(S)$ time. The inefficiency in the array implementation is due to the $\Theta(S)$-time methods.

The ***list implementation*** represents a `prof` R as a linked list $R.\ell\ell$ that logs updates to the elements stored in R. The linked list $R.\ell\ell$ is a singly linked list with a head and a tail pointer to support $\Theta(1)$-time concatenation. The INIT, UPDATE, and UNION functions are implemented using straightforward $\Theta(1)$-time linked list operations. The INIT method first deallocates any previous linked list, freeing the entries of $R.\ell\ell$ in $\Theta(1)$ amortized time, because each entry it frees must have been previously appended by UPDATE. Then INIT allocates an empty linked list with NULL head and tail pointers. Calling $\textsc{Update}(R, \langle s, v\rangle)$ appends a new linked-list element to $R.\ell\ell$ containing $\langle s, v\rangle$. Performing $\textsc{Union}(R, R')$ concatenates the linked lists $R.\ell\ell$ and $R'.\ell\ell$, and sets $R'.\ell\ell$ to an empty linked list. Similarly, PRINT operates in $\Theta(1)$ amortized time. The inefficiency in this implementation is space.

[2]Technically, the bound is $\Theta(1)$ expected time, because the implementation uses a hash table, but except for this one nit, the amortized bound better characterizes the performance of the data structure.

```
INIT($R$)
Free $R.arr$
Free $R.\ell\ell$
$R.\ell\ell = \emptyset$
$R.arr = \emptyset$
```

```
ASSIGN($R, R'$)
INIT($R$)
$R.\ell\ell = R'.\ell\ell$
$R.arr = R.arr$
$R'.\ell\ell = \emptyset$
$R'.arr = \emptyset$
```

```
_FLUSHLIST($R$)
if $R.arr == \emptyset$
    $R.arr =$ new Array($S$)
for $\langle s, v\rangle \in R.\ell\ell$
    $R.arr[h(s)]$ += $v$
Free $R.\ell\ell$
```

```
UPDATE($R, \langle s, v\rangle$)
if $R.arr \neq \emptyset$
    $R.arr[h(s)]$ += $v$
else APPEND($R.\ell\ell, \langle s, v\rangle$)
    if $|R.\ell\ell| == S$
        _FLUSHLIST($R$)
```

```
UNION($R, R'$)
if $R.arr \neq \emptyset$
    if $R'.arr \neq \emptyset$
        for $i = 0$ to $S - 1$
            $R.arr[i]$ += $R'.arr[i]$
        Free $R'.arr$
else $R.arr = R'.arr$
CONCATENATE($R.\ell\ell, R'.\ell\ell$)
if $|R.\ell\ell| \geq S$
    _FLUSHLIST($R$)
$R'.arr = \emptyset$
$R'.ll = \emptyset$
```

```
PRINT($R$)
_FLUSHLIST($R$)
for $i = 0$ to $S - 1$
    Output $R.arr[i]$
INIT($R$)
```

Figure 8-5: Pseudocode for the methods of the `prof` data structure, including a helper routine _FLUSHLIST. A `prof` R consists of a linked-list component $R.\ell\ell$ and an array component $R.arr$. The linked list $R.\ell\ell$ is a singly linked list with a cardinality field to keep track of the number of elements in the list and a head and tail pointer to enable $\Theta(1)$-time list concatenation.

Because every call to UPDATE allocates space for an update, the linked list uses space proportional to the total number of updates, which, for a Cilk computation with work T_1, is $\Theta(T_1)$ space.

The ***hybrid implementation*** that Cilkprof actually uses represents a `prof` R using both an array $R.arr$ and a linked list $R.\ell\ell$. Figure 8-5 gives the pseudocode for the `prof` methods. Conceptually, UPDATE and UNION use the linked list $R.\ell\ell$ to handle elements until $R.\ell\ell$ contains at least S updates. At this point, the elements in $R.\ell\ell$ are updated into the array $R.arr$, the linked list $R.\ell\ell$ is emptied, and UPDATE and UNION use the array $R.arr$ to handle future operations.

Intuitively, by combining the linked-list and array implementations, the `prof` data structure R enjoys the time efficiency of the linked list implementation with the space efficiency of the array implementation. Because UPDATE and UNION move elements from $R.\ell\ell$ into $R.arr$ when $R.\ell\ell$ contains at least S elements, R occupies $O(S)$ space. By initially storing elements in a linked list, the `prof` data structure can avoid performing an expensive UNION operation until it can amortize that expense against the elements that have been inserted. The following theorem formalizes this intuition.

Theorem 58 *The* `prof` *data structure uses at most $\Theta(S)$ space and supports each of* INIT, ASSIGN, UPDATE, UNION, *and* PRINT *in $\Theta(1)$ amortized time.*

PROOF. The time bound follows from an amortized analysis carried out using the accounting method [100, Ch. 17]. The amortization maintains the following invariants.

Invariant 59 *Each linked-list element carries* 2 *tokens of amortized time.*

Invariant 60 *Each array A carries $|A|$ tokens of amortized time.*

We analyze each of the `prof` methods in turn.

A call to $\textsc{Init}(R)$ takes $\Theta(1)$ time to free $R.\mathit{arr}$ and spends 1 token on each element in $R.\ell\ell$ to cover the cost of freeing that element. Then $\textsc{Init}$ performs $\Theta(1)$ operations in $\Theta(1)$ time to reinitialize the data structure, for a total of $\Theta(1)$ amortized time.

A call to $\textsc{Assign}$ is $\Theta(1)$ time plus a call to $\textsc{Init}$, for a total amortized time of $\Theta(1)$.

The helper routine $\_\textsc{FlushList}(R)$ is called only when its linked list $R.\ell\ell$ attains at least S elements. The routine may spend $\Theta(S)$ time to create a new array of size S, the entries of which are initialized to 0. This routine can use 1 token from each element in $R.\ell\ell$ to transfer that element's update to $R.\mathit{arr}$, free that element, transfer the element's other token to $R.\mathit{arr}$, and cover the $\Theta(1)$ real cost to initialize one entry of $R.\mathit{arr}$. Consequently, $\_\textsc{FlushList}$ takes $\Theta(1)$ amortized time and produces an array $R.\mathit{arr}$ with $|R.\ell\ell| \geq S = |R.\mathit{arr}|$ tokens, maintaining Invariant 60.

A call to $\textsc{Update}(R, \langle s, v\rangle)$ exhibits one of three behaviors. First, if the call executes line 37, then it takes $\Theta(1)$ real time. Otherwise, the call executes line 38, which is charged $\Theta(1)$ real time plus 2 amortized time units to append a new linked-list element with 2 tokens onto $R.\ell\ell$ while maintaining Invariant 59. At this point, if $|R.\ell\ell| = S$, then line 40 calls $\_\textsc{FlushList}$, which costs $\Theta(1)$ amortized time. Thus, $\textsc{Update}$ takes $\Theta(1)$ amortized time in every case.

A call to $\textsc{Union}(R, R')$ uses the tokens on $R'.\mathit{arr}$ to achieve a $\Theta(1)$ amortized running time. If the call executes lines 43–45, then each iteration charges 1 token from $R'.\mathit{arr}$ to cover the $\Theta(1)$ real cost to update an entry in $R.\mathit{arr}$ with an entry in $R'.\mathit{arr}$. Lines 43–45 therefore take $\Theta(1)$ amortized time. Line 47 takes $\Theta(1)$ time to concatenate two linked lists, and the analysis of lines 48–49 corresponds to that for lines 39–40 of $\textsc{Update}$. The amortized cost of $\textsc{Union}$ is therefore $\Theta(1)$.

A call to $\textsc{Print}(R)$ executes $\_\textsc{FlushList}$ in line 52 in $\Theta(1)$ amortized time, and spends the S available tokens in $R.\mathit{arr}$ to pay for sequencing through all the elements of R. Adding in the cost to call $\textsc{Init}$ in line 55 gives $\Theta(1)$ total amortized cost of printing.

The space bound on a `prof` data structure R follows from observing that the array $R.\mathit{arr}$ occupies $\Theta(S)$ space, and only line 38 in $\textsc{Update}$ and line 47 in $\textsc{Union}$ increase the size of the linked-list $R.\ell\ell$. Because lines 39–40 in $\textsc{Update}$ and lines 48–49 in $\textsc{Union}$ move the elements of $R.\ell\ell$ into $R.\mathit{arr}$ once the size of $R.\ell\ell$ is at least S, the linked list $R.\ell\ell$ never contains more than $2S$ elements, and R therefore occupies $O(S)$ total space. □

Discovering call sites dynamically

Let us now remove the assumption that the number S of call sites is known *a priori*. To handle call sites discovered dynamically as the execution unfolds, Cilkprof tracks the number S of unique call sites encountered so far. Cilkprof maintains the global hash table h using table doubling [100, Sec. 17.4], which can resize the table as it grows while still providing amortized $\Theta(1)$ operations. When Cilkprof encounters a new call site s, it increments S and stores $h(s) = S - 1$, thereby mapping the new call site to the new value of $S - 1$.

We must also modify the helper function $\_\textsc{FlushList}$. First, line 31 must check whether the size of the existing array matches the current value of S, rather than simply checking if it exists. If a new array is allocated, in addition to the linked-list elements being transferred to the new array, the old array elements must also be transferred. At the end, the old array must be destroyed.

We must also modify the UPDATE and UNION methods to ensure that both UPDATE and UNION maintain the same invariants in their amortization as stated in the proof of Theorem 58. Thus, the changes do not affect the asymptotic complexity of the `prof` data structure. Specifically, line 36 in the pseudocode for UPDATE must be modified as in _FLUSHLIST to check whether the size of the existing array matches the current value of S. With this change, a call to UPDATE adds a record to the linked list whenever the array is too small, even if the array already stores some records. Lines 41–46 in the pseudocode for UNION must also be modified to copy the elements of the smaller array into the larger.

8.6 The profile

This section describes the profile that Cilkprof computes. Although Section 8.4 describes how Cilkprof can measure the work and span of each call site assuming the program contains no recursive functions, in fact, Cilkprof must handle recursive functions with care to avoid overcounting their work and span. We define the "top-call-site," "top-caller," and "local" measurements that Cilkprof accumulates for each call site, each of which we found to be easy to compute and useful for analyzing the contribution of that call site to the work and span of the overall program. We describe how to compute these measures.

A Cilkprof ***measurement*** for a call site s consists of the following values for a set of invocations of s:

- an ***execution count*** — the number of invocations of s accumulated in the profile;
- the ***call-site work*** — the sum of the work of those invocations;
- the ***call-site span*** — the sum of the spans of those invocations.

Cilkprof additionally computes the parallelism of s as the ratio of s's call-site work and call-site span.

If programs contained no recursive functions, Cilkprof could simply aggregate all executions of each call site, but generally, it must avoid overcounting the call-site work and call-site span of recursive functions. Of the many ways that Cilkprof might accommodate recursive functions, we have found three sets of measurements of a call site, called the "top-call-site," "top-caller," and "local" measurements, to be particularly useful for analyzing the scalability of Cilk computations. These measurements maintain the basic algorithm's performance bounds given in Theorems 57 and 58 while also handling recursion.

Top-call-site measurements

A "top-call-site" measurement aggregates how much of the entire work or span of the computation can be attributed to a call site. Conceptually, the "top-call-site" measurement for a call site s aggregates the work and span of every execution of s that is not a recursive execution of s. Formally, an executed call site $xs \in XI$ is a ***top-call-site invocation*** if no executed call site $xi \in XI$ exists such that

$$xs \in \text{Trace}(xi) \wedge \sigma(xs) = \sigma(xi) \ .$$

Cilkprof's ***top-call-site measurement*** for s aggregates all top-call-site invocations of s.

(a)

```
38 int fib(int n) {
39   if (n < 2) return n;
40   int x, y;
41   x = cilk_spawn fib(n-1);
42   y = fib(n-2);
43   cilk_sync;
44   return (x + y);
45 }
46
47 int main(int argc, char *argv[]) {
48   int n, result;
49   // parse arguments
50   result = fib(n);
51   return 0;
52 }
```

(b)

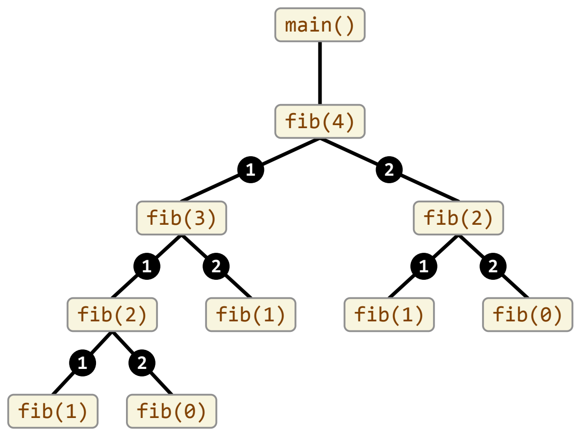


Figure 8-6: Cilk pseudocode and example invocation tree for a recursive program to compute Fibonacci numbers. **(a)** Cilk pseudocode for the program. **(b)** An invocation tree from running the program in (a) to compute the 4th Fibonacci number. Each rounded rectangle denotes a function instantiation, and an edge between tow rounded rectangles denotes the upper instantiation invoking the lower. The circled labels 1 and 2 on edges identify the call sites on lines 41 and 42, respectively, in the code in (a).

	Top-call-site			*Top-caller*		
Line	T_1	T_∞	T_1/T_∞	T_1	T_∞	T_1/T_∞
41	450,321,639	113,267	3,975.8	279,094,680	39,643	7,040.2
42	450,307,915	250,302	1,799.1	171,229,726	14,688	11,657.8
50	450,325,186	40,331	11,165.7	450,325,186	40,331	11,165.7

Figure 8-7: Top-call-site and top-caller work and span values in the on-work profile Cilkprof collects for running the recursive Fibonacci program in Figure 8-6 to compute `fib(30)`. All times are measured in nanoseconds.

An executed call site can be identified as a top-call-site invocation from the computation's invocation tree. Consider the invocation tree in Figure 8-6(b) for the parallel recursive Cilk program in Figure 8-6(a). Each edge in this tree corresponds to an ***invocation*** — an executed call site that either calls or spawns a child — and the labels on edges denote the corresponding call site. From Figure 8-6(b), we see that the executed call site spawning `fib(3)` is a top-call-site invocation, because no other execution of line 41 appears above `fib(3)` in the invocation tree. The spawning of `fib(2)` by `fib(3)` is not a top-call-site invocation, however, because both `fib(2)` and `fib(3)` above it in the tree are invoked from the same call site.

Cilkprof's top-call-site measurements are useful for assessing the parallelism of each call site. For a call site s, the ratio of the call-site work over the call-site span from the top-call-site data of s gives the parallelism of all nonrecursive executions of s in the computation, as if the computation performed each such call-site execution in series. This parallelism value can be particularly helpful for measuring the parallelism of executed call sites that occur on the critical path of the computation.

In a function containing multiple recursive calls, however, such as the `fib` routine in Figure 8-6(a), the top-call-site measurements are less useful for comparing the relative contribution of different call sites. For example, consider the top-call-site work and span values in Figure 8-7, which Cilkprof collected from running the code in Figure 8-6(a). As Figure 8-7 shows, the top-call-site work values for the recursive `fib` invocations on line 41 and line 42 are similar to that of the call to `fib` on line 50, and the top-call-site span values of these recursive invocations exceed that of line 50.

These large top-call-site measurements occur because the measurement aggregates multiple top-call-site executions of a call site under the same top-level call to `fib`. For example, as the invocation tree in Figure 8-6(b) shows, the invocations of `fib(2)` from `fib(4)`, `fib(1)` from `fib(3)`, and `fib(0)` from `fib(2)` under `fib(3)` are all top-call-site invocations for line 42. Similarly, the invocations of `fib(3)` from `fib(4)` and of `fib(1)` from `fib(2)` under `fib(4)` are both top-call-site invocations for line 41.

Top-caller measurements

"Top-caller" measurements aim to measure the relative contribution of different call sites in the same function to that function's work and span. In contrast to top-call-site measurements, the "top-caller" measurement for a call site s conceptually aggregates the work and span of every execution of s from a nonrecursive invocation of its caller. Formally, an executed call site $xs \in XI$ is a ***top-caller invocation*** if no executed call site $xi \in XI$ exists such that

$$xs \in \text{Trace}(xi) \wedge \varphi(\sigma(xs)) = \varphi(\sigma(xi)) .$$

Cilkprof's ***top-caller measurement*** for a call site s aggregates all top-caller invocations of s.

Like top-call-site invocations, top-caller invocations can be identified from the computation's invocation tree. Once again, consider the code in Figure 8-6(a) and its example invocation tree in Figure 8-6(b). The invocation producing `fib(3)` is a top-caller invocation, because no instantiation of `fib` exists above `fib(4)`, the function instantiation containing this invocation, in the tree. The invocation producing `fib(1)` under the right child of `fib(4)` is not a top-caller invocation, however, because between `fib(1)` and `fib(4)` is an instantiation of `fib`, namely, the instantiation of `fib(2)`. The top-caller invocations that occur in the invocation tree in Figure 8-6(b), therefore, are from `main()` to `fib(4)` and from `fib(4)` to each of its children.

Top-caller measurements can be useful for comparing call sites in the same function. The top-caller measurements in Figure 8-7, for example, show that the ratio of the aggregate work of the top-caller invocations of lines 41 and 42 is $279\,094\,680/171\,229\,726 \approx 1.63$, which is approximately the golden ratio $\phi = (1+\sqrt{5})/2 \approx 1.61$. This relationship makes sense, because `fib(n)` theoretically incurs $\Theta(\phi^n)$ work.

The top-caller measurements provide no information for call sites that are never reached from a top-level instantiation of a function, however. Consider the divide-and-conquer matrix-multiplication program in Figure 8-8(a) and its invocation tree illustrated in Figure 8-8(b). As Figure 8-8(b) shows, the function `base` called in the base case of `mm` is never called by the top-caller invocation of `mm` from `main`. Consequently, as the top-caller measurements in Figure 8-9 for this program show, Cilkprof measures the top-caller call-site work and span values for `base` to be 0. From these top-caller values, one cannot conclude that most of the computation of `mm`, in fact, occurs under calls to `base`.

(a)

```
void mm(double *C, double *A, double *B,
        size_t dim, size_t n) {
  if (n < COARSENING)
    return base(C, A, B, dim, n);
#define X(M,r,c) (M + (r * dim + c)*(n/2))
  cilk_spawn mm(X(C,0,0), X(A,0,0), X(B,0,0),
                dim, n/2);
  cilk_spawn mm(X(C,0,1), X(A,0,0), X(B,0,1),
                dim, n/2);
  cilk_spawn mm(X(C,1,0), X(A,1,0), X(B,0,0),
                dim, n/2);
             mm(X(C,1,1), X(A,1,0), X(B,0,1),
                dim, n/2);
  cilk_sync;

  cilk_spawn mm(X(C,0,0), X(A,0,1), X(B,1,0),
                dim, n/2);
  cilk_spawn mm(X(C,0,1), X(A,0,1), X(B,1,1),
                dim, n/2);
  cilk_spawn mm(X(C,1,0), X(A,1,1), X(B,1,0),
                dim, n/2);
             mm(X(C,1,1), X(A,1,1), X(B,1,1),
                dim, n/2);
  cilk_sync;
}

int main(int argc, char *argv[]) {
  double *C, *A, *B;
  int n;
  // parse arguments
  // initialize C, A, and B
  mm(C, A, B, n, n);
  return 0;
}
```

(b)

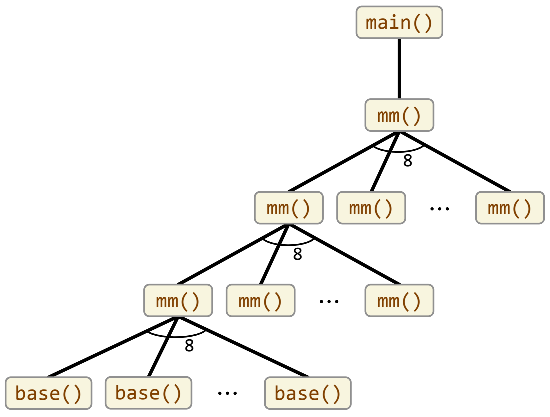


Figure 8-8: Cilk pseudocode and an example invocation tree for a divide-and-conquer parallel matrix-multiplication program. **(a)** Cilk pseudocode for the matrix-multiplication program. The recursive mm routine in this program calls the function base in its base case. The variable COARSENING is a fixed constant defining the maximum size of matrices to multiply in the base case. **(b)** An invocation tree for the matrix-multiplication program in (a). Each rectangle denotes a function instantiation, and an edge from one rectangle to a rectangle below it denotes the upper invocation calling the lower.

Local measurements

Cilkprof also computes "local" work and span measurements for each call site, which exclude child functions spawned or called from that call site. The ***local*** measurement for a call site aggregates a "local work" and a "local span" for every execution of that call site. The ***local work*** of an executed call site $xs \in XI$ is the work in Trace(xs) minus the work in the traces of the functions that xs invokes. By ignoring the contributions of its children, the local work and local span values for all executions of a call site can be aggregated without overcounting executed instructions in recursive calls.

The local measurements call sites are often useful for examining functions invoked in the base case of a recursive routine. The local measurements in Figure 8-10 for the mm program, for example, make it clear that most of the total work of the call to mm from main occurs in calls to base. Furthermore, as we observed in the quicksort example in Section 8.2, these

	Top-call-site			*Top-caller*		
Line	T_1	T_∞	T_1/T_∞	T_1	T_∞	T_1/T_∞
56	185,830,187	185,830,187	1.0	0	0	—
58	77,417,401	22,010,873	3.5	23,473,633	405,947	57.8
60	77,475,639	21,983,150	3.5	23,384,174	411,099	56.9
62	77,440,990	21,988,390	3.5	23,403,194	402,281	58.2
64	77,262,499	21,853,710	3.5	23,378,967	387,880	60.3
68	77,400,652	21,966,823	3.5	23,384,823	391,066	59.8
70	77,429,530	21,945,602	3.5	23,362,153	387,466	60.3
72	77,330,287	21,890,345	3.5	23,390,504	389,726	60.0
74	77,241,198	21,834,699	3.5	23,371,773	388,095	60.2
84	187,150,784	803,122	233.0	187,150,784	803,122	233.0

Figure 8-9: Top-call-site and top-caller work and span values in the on-work profile Cilkprof produces for running the divide-and-conquer matrix multiplication code in Figure 8-8 to multiply two 512×512 matrices of doubles. All times are measured in nanoseconds.

Line	T_1	T_∞	T_1/T_∞
56	185,830,187	185,830,187	1.0
58	195,079	177,786	1.1
60	183,232	168,249	1.1
62	187,472	169,541	1.1
64	110,374	97,656	1.1
68	181,666	161,616	1.1
70	177,803	163,037	1.1
72	174,297	155,647	1.1
74	109,111	96,262	1.1
84	1,563	957	1.6

Figure 8-10: Local work and span values in the on-work profile Cilkprof produces for running the divide-and-conquer matrix multiplication code in Figure 8-8 to multiply two 512×512 matrices of doubles. All times are measured in nanoseconds.

local work and span values can effectively identify which functions contribute directly to the span of the computation.

Local measurements can also provide a breakdown of the work or span of a top-caller invocation that complements the breakdown provided by top-caller measurements. The local work measurements in Figure 8-10 for the `mm` program, for example, sum to the top-caller work of line 84. Furthermore, the sum of the top-caller work values in Figure 8-9 for the call sites in `mm` (lines 56–74) plus the local work measurement for line 84 gives the top-caller work of line 84. These two breakdowns complement each other. While the breakdown by top-caller work demonstrates that the work of line 84 is distributed roughly evenly between its recursive calls (lines 58–74), the breakdown by local work demonstrates that the majority of the work under line 84 lies in its base case.

The local parallelism of a call site s — the ratio of the local work and local span of s — does not accurately reflect the parallelism of s, however. As Figure 8-10 shows, by excluding the work and span contributions of each executed call site's children, most local-parallelism values are close to 1, even for call sites such as line 84 which exhibit ample parallelism.

The top-call-site, top-caller, and local measurements for a call site s each measure qualitatively different things about s. These measurements provide different, complementary perspectives on the work and span of a Cilk program, and they each seem to be useful for

CPU	Intel Xeon E5-2665
Clock	2.4 GHz
Hyperthreading	Disabled
Turbo Boost	Enabled
Cores per processor chip	8
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	256 KiB
L3 cache/socket	20 MiB
DRAM	32 GiB
Operating system	Fedora 16, custom Linux kernel 3.6.11

Figure 8-11: Technical specifications of the machine used for benchmarking. The kernel was patched with support for thread-local memory mapping used in Cilk-M [238]. This patch was irrelevant to the experiment, and we do not believe it affects the numbers.

analyzing a program in different ways. An interesting open question is whether there are other measurements that are as useful as these three for diagnosing scalability bottlenecks.

8.7 Empirical evaluation

To implement Cilkprof, we modified a branch [195] of the LLVM [232] compiler that supports Cilk Plus [196] to instrument function entries and exits, as well as calls into the Cilk runtime from the program for handling `cilk_spawn` and `cilk_sync` statements. A Cilk program is compiled with the modified compiler to produce a binary executable that executes the Cilkprof algorithm as a shadow computation. On a suite of 16 benchmark programs, we compared the Cilkprof running time of each benchmark with the benchmark's serial running time compiled with the unmodified compiler, both executions using optimization level `-O3`. Compared with this "native" serial execution, Cilkprof incurs a geometric-mean multiplicative slowdown of 1.9 and a maximum slowdown of 7.4.

Results

We compared the running time of Cilkprof on each benchmark to the native serial running time of the benchmark, that is, the running time of the benchmark when compiled with no instrumentation. Figure 8-11 summarizes the specifications of the benchmarking machine used for our experiments.

To study the empirical overhead of Cilkprof, we compiled a suite of 16 application benchmarks, as Figure 8-12 describes. The `mm`, `quicksort`, and `fib` benchmarks correspond to the Cilk pseudocode in Figures 8-8, 8-1 and 8-6, respectively. The `pbfs` benchmark is a parallel breadth-first search code that implements the PBFS algorithm of Chapter 3. We converted the `dedup` and `ferret` benchmarks from the PARSEC benchmark [45, 46] to use Cilk linguistics and a `reducer_ostream` (which is part of Cilk Plus) for writing output deterministically. The `leiserchess` program performs a parallel speculative game-tree search using Cilk. The `hevc` benchmark is a 30,000-line implementation of the H265 video encoder and decoder [386] that we parallelized using Cilk. The remaining benchmarks are the same benchmarks included in the Cilk-5 distribution [146].

Figure 8-12 presents our empirical results. As the figure shows, the Cilkprof implementation incurs a geometric mean slowdown of $1.9\times$ on these benchmarks compared to

Benchmark	*Description*	*Input size*	*Overhead*
`mm`	Square matrix multiplication	2048×2048 matrix	0.99
`dedup`	Compression program	large	1.03
`lu`	LU matrix decomposition	2048×2048 matrix	1.04
`strassen`	Strassen matrix multiplication	2048×2048 matrix	1.06
`heat`	Heat diffusion stencil	$4096 \times 1024 \times 40$ spacetime	1.07
`cilksort`	Parallel mergesort	10 M elements	1.08
`pbfs`	Parallel breadth-first search	$\lvert V \rvert = 8\,\mathrm{M}$, $\lvert E \rvert = 55.8\,\mathrm{M}$	1.10
`fft`	Fast Fourier transform	8 388 608	1.15
`quicksort`	Parallel quicksort	100 M elements	1.20
`nqueens`	n-Queens problem	12×12 board	1.27
`ferret`	Image similarity search	large	2.04
`leiserchess`	Speculative game-tree search	5.8 M nodes	3.72
`collision`	Collision detection in $3D$	528 032 faces	4.37
`cholesky`	Cholesky decomposition	$2\,\mathrm{k} \times 2\,\mathrm{k}$ matrix, 16 k nonzeros	4.54
`hevc`	HEVC video encoding and decoding	5 frames	6.25
`fib`	Recursive Fibonacci	35	7.36

Figure 8-12: Application benchmarks demonstrating the performance overhead of the Cilkprof prototype tool. The benchmarks are sorted in order of increasing overhead. For each benchmark, the *Overhead* column gives the ratio of its running time with when compiled with the Cilkprof implementation over its running time without instrumentation. Each ratio is computed as the geometric mean ratio of 5 runs with Cilkprof and 5 runs without Cilkprof. We used a modified version of the Cilk Plus/LLVM compiler to compile each benchmark with Cilkprof, and we used the original version of the Cilk Plus/LLVM compiler to compile the benchmarks with no instrumentation. The Cilkprof implementation and benchmark codes were compiled using the `-O3` optimization level.

the uninstrumented version of the benchmark. Furthermore, the maximum multiplicative overhead we observed on any benchmark was 7.4.

Optimizations

Our Cilkprof implementation contains several optimizations:

- For basic timing measurements, we chose to use a cycle counter to measure blocks of instructions, rather than naively incrementing a counter for every instruction executed, as in the basic pseudocode from Figure 8-4. We also adjust the measured numbers to compensate for the time it takes Cilkprof to execute the instrumentation.
- For a function F with no `cilk_spawn`'s, the implementation maintains only the prefix span variable $F.p$ to store the span of F, rather than all 3 span variables.
- If a function G calls F, then the implementation sets the `prof` data structure for $F.p$ to be the `prof` data structure for either $G.p$, if G has no outstanding spawned children when it calls F, or $G.c$, otherwise.
- When the span variable associated with a `prof` data structure R is set to 0, the implementation simply clears the nonempty entries of the array $R.arr$, rather than freeing its memory.
- The implementation maintains the set of nonempty entries in each `prof` data structure array in order to optimize the processes of combining and clearing those entries.

These optimizations together reduced Cilkprof's overhead on the `fib` benchmark by a factor of 5 and its overhead on the `leiserchess` benchmark dropped by a factor of 9.

At the risk of losing some information, Cilkprof declines to instrument inlined functions, which, in Intel Cilk Plus, cannot spawn. To do so, we modified the compiler such that, when it inlines a function, it removes the instrumentation for the inlined version of that function. When this program runs with Cilkprof, therefore, the work of the inlined function influences the work and span of its parent, but it will not create a separate entry in the profile Cilkprof produces.

We feel that this optimization is reasonable because inlined functions are typically unlikely to be scalability bottlenecks on their own. For example, the compiler often inlines C++ object methods to extract or set fields of that object, which the programmer wrote to provide a convenient abstraction in the program code, but which the compiler can often implement with a handful of data movement instructions in the caller. If Cilkprof instruments such a function, then Cilkprof incurs overheads to measure and record *very* few instructions. The cost of instrumenting such functions therefore seems to outweigh the benefits to scalability analysis.

We examined Cilkprof's overhead when inlined functions are instrumented on the application benchmarks. For all benchmarks except `lu`, `leiserchess`, `collision`, and `hevc`, Cilkprof incurred less than 2 times the overhead it incurred when inlined functions in that benchmark were not instrumented. On the `leiserchess` and `collision` benchmarks, however, instrumenting inlined functions increased Cilkprof's overhead by a factor of 8 to 10. Both of these benchmarks make extensive use of small functions that simply get or set fields of an object, which are particularly light weight when inlined. Although this optimization does not affect every benchmark, it can dramatically improve Cilkprof's performance on the benchmarks it does affect.

8.8 Case study: PBFS

One of our first successes with Cilkprof[3] came when diagnosing a parallelism bottleneck in PBFS, the 1800-line parallel breadth-first search Cilk program described in Chapter 3. After just 2 hours of work using Cilkprof, we were able to identify a parallelism bottleneck in the PBFS code. Fixing this bottleneck enhanced the parallelism of the code by a factor of about 5. This section presents our experience diagnosing a scalability bottleneck in the PBFS code. You should not need to understand either the PBFS algorithm or its implementation to follow this case study.

After designing and building PBFS, we observed that the code failed to achieve linear speedup on 8 processors. For example, PBFS was achieving a parallel speedup of 4 to 5 on our Grid3D200 benchmark graph, a 7-point finite-difference mesh generated using the Matlab Mesh Partitioning and Graph Separator Toolbox [159], on which PBFS explored 8M vertices and 55.8M edges during a search of depth 598. A back-of-the-envelope calculation suggested that the measured parallelism of PBFS should be around $200 - 400$, ample for 8 processors, if one follows the rule of thumb that a program should have at least 10 times more parallelism than the number of processors for scheduling overhead to be negligible.

We suspected that the scalability of this PBFS code suffered from insufficient memory bandwidth on the machine. For example, when we artificially inflated the amount of computation that the code performed in the base case of its recursive helper functions, then the code did exhibit linear speedup. The problem with this test, however, is that it also increased the parallelism of the code. We ran Cilkview on the original PBFS code to en-

[3]Actually, the original case study used the (much slower) Cilkprof Pintool [193] we built in collaboration with Intel. Since the data from this early experiment have since been lost, we recreated the experiment with our LLVM-based Cilkprof implementation.

sure that insufficient parallelism was not the issue. Cilkview, however, reported that the parallelism of this PBFS code was merely 12, which is not ample parallelism for 8 processors.

We ran the PBFS code with Cilkprof and examined Cilkprof's profiles. The on-work profile showed us that the call to `pbfs` — our parallel BFS routine — from `main` accounted for most of the work of the program, and that the parallelism of `pbfs` was small, just as Cilkview had found. To discover what methods contributed most to the span, we sorted Cilkprof's data by decreasing local T_∞ on span. Viewing the data from this perspective showed us that the following three methods contributed the most to the span of the program overall:

1. First was a call to `parseBinaryFile`, a serial function that parses the input graph.
2. Second was a call to the serial `Graph` constructor to create the internal data structure storing the graph from the input.
3. Third was a call to `pbfs_proc_Node`, a function that processes a constant-sized array of graph vertices.

Although the top two entries were not called from `pbfs`, the third entry for `pbfs_proc_Node` was called in the base case of the recursive helper methods of `pbfs`. Comparing the local T_∞ on span of `pbfs_proc_Node` to the top-caller T_∞ of `bfs` showed us that this method accounted for 66 % of the span of `bfs`. Furthermore, the top-call-site parallelism values from Cilkprof showed us that all invocations `pbfs_proc_Node` were serial.

These data led us to look more closely at `pbfs_proc_Node`. We discovered that this method evaluates a constant-sized array of vertices in the graph. Because the input array has constant size, this method evaluated the contents of this array serially. In the code, however, the size of this array was tuned to optimize the insertion of vertices into the array. The constant size of this array was therefore too large for `pbfs_proc_Node`, causing the serial execution of `pbfs_proc_Node` to become a scalability bottleneck.

We parallelized the `pbfs_proc_Node` function to process its input array in parallel with an appropriate base-case size. We then ran our modified PBFS code through Cilkprof and sorted the new data by local T_∞ on span to examine the effect of our efforts. We found that, although `pbfs_proc_Node` was still the third-largest contributor to the span of the program, the local T_∞ on span is a factor of 6 larger. Furthermore, the parallelism of `pbfs` is now 60, a factor of 5 larger than its previous value. Finally, `pbfs_proc_Node` accounts for 48 % of this span. We also confirmed that reducing the new base-case size of `pbfs_proc_Node` can increase the parallelism of `pbfs` to 100, at the cost of scheduling overhead.

8.9 Related work

This section reviews related work on performance tools for parallel programming.

We chose to implement Cilkprof using compiler instrumentation (e.g., [350, 351]), but there are other strategies we could have used to examine the behavior of a computation, such as asynchronous sampling (e.g., [171]) and ***binary instrumentation*** (e.g., [72,129,267, 296]). Although asynchronous sampling provides low-overhead solutions for some analytical tools, we do not know of a way to measure the span of a multithreaded Cilk computation by sampling. Cilkview [180] is implemented using the Pin binary-instrumentation framework [267] augmented by support in the Intel Cilk Plus compiler [198] for low-overhead annotations [192], and we collaborated with Intel to build a prototype Cilkprof as a Pintool. Because we found that this prototype Cilkprof ran slowly, we chose to implement Cilkprof using compiler instrumentation in order to improve its performance. In fact, because it uses compiler instrumentation, the Cilkprof implementation outperforms the existing Cilkview implementation, which only computes work and span for the entire computation and does not produce profiles of work and span for every call site as Cilkprof does.

Many parallel performance tools examine a parallel computation and report performance characteristics specific to that architecture and execution. Tools like HPCToolkit [3], Intel VTune Amplifier [199], and others [74, 219, 355] measure system counters and events, and provide reports based on a program execution. HPCToolkit, in particular, is an integrated suite of tools to measure and analyze program performance that sets a high standard for capability and usability. HPCToolkit uses statistical sampling of timers and hardware performance counters to measure a program's resource consumption, and attributes measurements to full calling contexts.

Other approaches for identifying scalability bottlenecks include normalized processor time [17] or the more precise parallel idleness metric [383]. The idea is that, in a work-stealing concurrency platform, if at some particular point in time some worker threads are idle, then we can assign blame to the function that is running on the other workers: if that function were more parallel, then the idle threads would be doing something useful. These are helpful metrics for identifying bottlenecks on the current architecture, and answer the question as to whether the program, run on a P-processor machine has at least P-fold parallelism. But they don't provide scalability analysis beyond P processors.

In contrast to all of these applications and approaches, Cilkprof's analysis applies to the measured work and span. Work and span are good metrics for inferring bounds on parallel speedup on architectures with any number of processors. A program compiled for Cilkprof will generate profile information that is generally applicable, rather than just for the architecture on which it was run. Additionally, Cilkprof is distinguished in that it uses direct instrumentation rather than statistical sampling.

Whereas Cilkprof computes the parallelism of call sites in a parallel program, the Kremlin [149, 207] and Kismet [206] tools analyze serial programs to suggest parallelism opportunities and to predict the impact of parallelization. Kremlin can suggest which parts of a serial program might benefit from parallelization. It estimates the parallelism of a serial program using "hierarchical critical-path analysis" and connects to a "parallelism planner" to evaluate many possible parallelizations of the program. Based on its determination of which regions (loops and functions) of the program should be parallelized, it computes a work-span profile of the program, computing a "self-parallelism" metric for each region, which estimates the parallelism that can be obtained from parallelizing that region separate from other regions. The analysis produces a textual report as output suggesting which regions should be parallelized. Kismet, which is a product of the same research group, attempts to predict the actual speedup after parallelization, given a target machine and runtime system.

8.10 Conclusion

Cilkprof enables performance engineers to profile the scalability of a Cilk program, just as they might profile the running time of a serial algorithm. Cilkprof thereby supports a principled approach to tracking down serial bottlenecks in Cilk codes. Our work on Cilkprof has left us with some interesting research questions. We conclude by addressing issues of Cilkprof's user interface, parallelizing Cilkprof, and making Cilkprof functionally more "complete."

Chief among the open issues is user interface. How should the profiles produced by

Cilkprof be communicated to a Cilk programmer? Although we ourselves used just a spreadsheet to divine important scalability properties of PBFS, for example, we do not recommend this method to others. A good UI integrated with the development environment would make diagnosing scalability issues much easier for average programmers.

Even though Cilkprof analyzes parallel programs, it still runs them serially. As the number of processors grows, it becomes less and less acceptable to resort to a serial execution. In principle, nothing precludes Cilkprof from running in parallel, but we have thus far been unable to create a provably good algorithm. One problem is that amortization plays havoc with the critical path of a parallel program. At some cost in programming complexity, we could deamortize the `prof` data structures, but it is also tricky to parallelize the strategies for handling recursion.

Cilkprof offers many opportunities for functional enhancements. The on-work and on-span profiles seem natural enough, but maybe there are better alternatives to top-call-site, top-caller, and local measurements. In addition, Cilkprof computes on-span profiles only for call sites that lie on the global critical path. Sometimes, the critical path of a computation can be qualitatively different depending on the size of the program input. For example, two computations A and B are run in parallel, where the span of A is smaller than the span of B for small inputs, but the reverse is true for large inputs. Rather than run at scale, it could be more productive if Cilkprof were to report span-on-span profiles not just for the global critical path, but for all critical paths within all functions. Although the space required might be quadratic in call sites, such a profile would greatly speed detective work, and the cross-product of sites might be considerably sparse for many programs. Unfortunately, we do not yet see a way to calculate such a profile without also blowing up the overheads significantly.

Chapter 9

Race Detection for Cilk Programs That Use Reducer Hyperobjects

This chapter presents the Rader race detector [240]. This work was conducted in collaboration with I-Ting Angelina Lee.

9.1 Introduction

A multithreaded Cilk program that is "ostensibly deterministic" can nevertheless behave nondeterministically due to programming errors in the code. Typically these errors, also called ***races***, occur when the program fails to coordinate parallel operations on a shared variable, causing accesses and updates to be performed on the variable in a nondeterministic order based on scheduling happenstance. Although provably efficient and correct race detection algorithms exist for Cilk computations[1] [37, 134, 197], they do not provide the same guarantees when the program under test employs a reducer hyperobject [144], an advanced linguistic feature supported in various Cilk dialects. Races involving the use of a reducer are particularly challenging to debug, because such races can expose the nondeterminism in how the Cilk runtime system manages a reducer. This chapter addresses the question of how to efficiently and correctly detect races in Cilk programs that use reducers.

Reducer hyperobjects [144], which are supported by Cilk dialects including Intel Cilk Plus [196], Cilk++ [246], and Cilk-M [238], provide a general reduction mechanism for Cilk programs and exhibit several useful properties.

- Reducers operate on arbitrary Cilk code. They are not tied to any particular linguistic construct.
- Reducers can operate on any abstract data type, including a set, a linked list, or even a *user-defined data type*, so long as the user supplies an appropriate REDUCE operator.
- To produce a deterministic result, a reducer's update and REDUCE operations do not need to be commutative; associativity suffices.

In contrast, other reduction mechanisms, such as OpenMP's reduction clause [306] or Microsoft's PPL's combinable objects [278], tie the reduction mechanism to a particular construct, such as a parallel loop, or require reductions to be commutative.

[1]Throughout this chapter, when we discuss a Cilk program, we shall mean the program with a given input. When we discuss a Cilk computation, we shall mean a particular execution of a Cilk program with a given input.

Although these properties make reducers a powerful general-purpose reduction mechanism, they leave open opportunities for programming errors that can produce races involving reducers. Such errors can, in particular, expose the nondeterminism in the Cilk runtime system's efficient management of reducers, which includes two significant optimizations [144]. First, a new reducer view is created only when a worker thread steals some parallel subcomputation. Second, views are reduced together in an opportunistic fashion, causing reductions to occur in a nondeterministic order. Consequently, the state of a reducer's view at a particular program point, the number of views created throughout the execution, and the partial order in which views are reduced together are all nondeterministic, depending on how the scheduling plays out. This nondeterminism is typically encapsulated by the reducers when used and programmed correctly, but it can become observable due to programming errors.

The incorrect use of a reducer gives rise to two unique types of races.

The first type of race, called a ***view-read race***, occurs when a Cilk computation reads the value of a reducer at a program point where the read might produce a nondeterministic value, such as before all previously spawned subcomputations that might update the reducer have necessarily returned. Because the Cilk runtime system creates and reduces views based on scheduling, such a read can cause multiple runs of the same Cilk program to produce different results.

The second type of race is a determinacy race, which occurs when two logically parallel instructions operate on the same memory location, and at least one of them is a write. Although ordinary Cilk programs can contain determinacy races, a Cilk program that uses a reducer can contain a determinacy race that involves a ***view-aware*** instruction — an instruction executed in creating, updating, or reducing views of a reducer. (In contrast, we refer to all other instructions that do not operate on views as ***view oblivious***.) Such a determinacy race is particularly challenging to debug, because a view-aware instruction involved in a race might not execute at all if the Cilk runtime system schedules the computation differently and thus manages the views differently.

Existing algorithms for detecting determinacy races in Cilk computations, including the SP-bags algorithm [134], the SP-order algorithm [37], and the SP-hybrid algorithm [37], do not support detecting races involving reducers. Extending these race detection algorithms to handle reducers while providing provable guarantees is non-trivial for two reasons. First, the use a reducer generates parallel control dependencies that violate the structural assumptions that these algorithms depend on. Specifically, the computation can no longer be modeled as a series-parallel dag [134], which is a property that existing algorithms rely on. Second, different runs of a Cilk program that uses a reducer can cause different view-aware instructions to be executed, depending how the scheduling plays out. Providing complete coverage could potentially require executing exponentially many different schedules to elicit all possible view-aware instructions. Consequently, existing tools that embody the SP-bags algorithm,[2] such as the Nondeterminator [134] and Cilkscreen [197], cannot guarantee correctness when one of the instructions involved in a race is executed to operate on a reducer view.

Contributions

In this chapter, we show how to efficiently and correctly detect these two types of races in a Cilk computation that uses reducers. Our race detection algorithms embed the programming and execution models of reducers into race-detection algorithms to locate races involving

[2]To the best of our knowledge, no implementation of the SP-order and SP-hybrid algorithms exists.

reducers in a provably effective and efficient manner. Consequently, these race-detection algorithms allow performance engineers to quickly zero in on programming bugs that involve reducers. Specifically, we make the following contributions.

The Peer-Set algorithm. We present the Peer-Set algorithm, which executes a Cilk computation serially and analyzes its logical parallelism to detect view-read races. The algorithm is provably correct, meaning it reports a view-read race if and only if the Cilk computation contains one. For a Cilk computation that runs in time T on one processor, the Peer-Set algorithm executes in time $O(T\alpha(v,v))$, where α is Tarjan's functional inverse of Ackermann's function [384], a very slowly growing function which, for all practical purposes, is bounded above by 4.

The SP+ algorithm. We present the SP+ algorithm, which detects determinacy races in Cilk computations that use reducers. The SP+ algorithm extends Feng and Leiserson's SP-bags algorithm [134] for detecting determinacy races in ordinary Cilk programs that do not employ reducers. The SP+ algorithm takes as input a Cilk program, its input, and a ***steal specification*** that effectively fixes the schedule. That is, a steal specification specifies the program points at which steals occur and which Reduce operations execute. Like the Peer-Set algorithm, SP+ executes the computation serially while it simulates the steals according to the steal specification to detect determinacy races. The SP+ algorithm is provably correct, meaning it reports a determinacy race in the computation if and only if one exists, regardless of whether that determinacy race occurs due to an operation on a reducer. Furthermore, the SP+ algorithm executes efficiently in time $O((T + M\tau)\alpha(v,v))$, where T is the work of the Cilk program on the given input excluding the runtime's work to create or reduce views, M is the number of steals in the steal specification, and τ is the worst-case running time of a Reduce operation. The SP+ algorithm thus incurs overhead over the SP-bags algorithm only to execute Reduce operations and simulate necessary steals.

Implementation and empirical evaluation of the algorithms. We have developed a prototype tool, called ***Rader***, that implements both the Peer-Set and SP+ algorithms to debug Cilk computations that use reducers. Rader implements the Peer-Set and SP+ algorithms by using compiler instrumentation to track memory accesses and parallel control dependencies. Using Rader, we empirically demonstrate the efficiency of both algorithms in practice. We ran Rader on six application benchmarks that use reducers. Compared to running each benchmark without instrumentation, Rader incurred geometric-mean multiplicative overheads of 2.56 and 16.94 to run the Peer-Set and SP+ algorithms, respectively.

Analysis of SP+'s coverage guarantees. We show how the SP+ algorithm can be used to efficiently check all executions of an "ostensibly deterministic" Cilk program for determinacy races that involve at least one view-oblivious instruction. A single run of the SP+ algorithm detects determinacy races in one possible schedule and thus has limited coverage; it elicits only a subset of all possible view-aware instructions. While this behavior is useful for debugging methodologies such as regression testing, a single run of SP+ does not guarantee that *no* execution of the program will exhibit a determinacy race. Although an exponential number of steal specifications exist for a given Cilk program, one can do better for most Cilk programs. Most Cilk programs are written to be ***ostensibly deterministic***, meaning that, in the absence of a race, its view-oblivious instructions are fixed across all executions regardless of scheduling, and it employs only reducers with semantically associative Reduce operations. For such Cilk programs, we show how to construct a polynomial number of steal specifications to elicit all possible view-aware instructions. The SP+ algorithm can use these steal specifications to exhaustively check for determinacy races between view-oblivious and view-aware instructions.

The remainder of this chapter is organized as follows. Section 9.2 presents an example program that illustrates how races involving reducers can arise. Sections 9.3 and 9.4 present the Peer-Set algorithm, along with intuition and a formal proof of its correctness. Section 9.5 presents the SP+ algorithm and a high-level intuition for its correctness. To formally argue for SP+'s correctness, Section 9.6 first introduce the "spawn parse tree" and the "view parse tree," two concepts that we use in the formal proof, and then Section 9.7 formally shows that SP+ algorithm correctly detects determinacy races. Section 9.8 shows that executing SP+ with polynomial number of different steal specifications is necessary and sufficient to elicit all possible view-aware instructions in a ostensibly deterministic Cilk program, thereby providing the stated coverage guarantees. Section 9.9 describes our prototype implementation of Rader and empirically evaluates its performance. Section 9.10 discusses related work and Section 9.11 provides concluding remarks.

9.2 Examples of races that involve a reducer

This section provides an motivational example to illustrate how races that involve operations on a reducer can occur. We walk through the example to illustrate how a subtle programming error can trigger a race between user code and a reduce operation on a reducer.

Chapter 2 describes the basics of how the Cilk runtime system supports reducers. To simplify the description of races on reducers, we shall assume that the reducer supports a single, serial Update method, which the program can invokes to modify the current view of the reducer. The race-detection techniques described in this chapter generalize to handle reducers with a variety of serial or parallel update operations. Formally, a view-aware strand occurs when the Cilk computation executes a Update, Create-Identity, or Reduce operation. Other strands in the Cilk computation are view-oblivious strands.

Example view-read race

To illustrate how a view-read race can occur, let us consider the code for the `update_list` routine shown in Figure 9-1. The function `update_list` takes in as parameters an integer `n` and a user-defined `list` of type `MyList` that implements a singly linked list with a head pointer and a tail pointer to enable fast list concatenation. The `update_list` routine spawns `foo` with `n` and `list_reducer` to perform some computation, which can execute in parallel with the continuation on lines 5–7, a parallel loop that inserts `n` elements into the linked list. To coordinate parallel accesses to the list, `update_list` wraps the given linked list in a reducer on line 2. Since the reducer has a user-defined view type, the programmer must also supply the functions for implementing the reducer's Create-Identity and Reduce operations, which are defined via the `list_monoid` type. (The actual implementation of `list_monoid` is not shown in the pseudocode.) Assuming that `list_monoid` implements these functions correctly, `update_list` does not itself contain a determinacy race, because the runtime coordinates parallel updates to the linked list via the use of a reducer.

```
01 void update_list(int n, MyList<int>& list) {
02   cilk::reducer< list_monoid<int> > list_reducer;
03   list_reducer.set_value(list);
04   int x = cilk_spawn foo(n, list_reducer);
05   cilk_for (int i = 0; i < n; ++i) {
06     list_reducer.view().insert(i);
07   }
08   // If run, the following commented-out statement
09   // would produce a view-read race.
10   // list = list_reducer.get_value();
11   cilk_sync;
12   list = list_reducer.get_value();
13 }
14
15 void race(int n, MyList<int>& list) {
16   int length = 0;
17   MyList<int> copy(list);
18   length = cilk_spawn scan_list(list);
19   update_list(n, copy);
20   cilk_sync;
21   return;
22 }
```

Figure 9-1: Example Cilk program that contains a determinacy race on the REDUCE operation of a custom linked-list reducer. The `MyList` type is a user-defined list type that implements a singly linked list with a head pointer and a tail pointer to enable fast list concatenation. The `list_monoid` type implements a reducer whose view type is `MyList`. The actual implementation of `list_monoid` is not shown.

As written, the routine does not exhibit a view-read race, because line 3 initializes the value of `list_reducer` before anything is spawned, and line 12 retrieves the value of `list_reducer` after all spawned subcomputations that can use the reducer have returned. If, however, the commented-out call to `get_value` on line 10 were executed, then the code would contain a view-read race, because `foo` might be accessing the reducer in parallel at that point.

Example determinacy race

To illustrate how a determinacy race involving a reducer can occur, let us consider the code for `race` in Figure 9-1, which calls the `update_list` routine. In this code, the `race` routine invokes `scan_list`, which iterates through the elements of `list` until one is found with a null pointer to the next element. Because `scan_list` is spawned, it can run in parallel with its continuation on line 19, which calls `update_list`. Because `update_list` might actually insert into the list, the `race` routine makes a copy of the list first at line 17 and passes the copy to `update_list`, so as to allow `scan_list` to scan the snapshot of the list without the new inserts performed by `update_list`.

Despite this precaution, this code contains a determinacy race, assuming the programmer did not implement a custom copy constructor for `MyList`. The default copy constructor for `MyList`, which line 17 invokes, only performs a shallow copy. As a result, even though `copy` is a new `MyList` object, created with its own distinct head and tail pointers, both `list` and `copy` still point to the same set of linked-list elements, leading to a determinacy race in the code. In particular, whenever `scan_list` reaches the last linked-list node of `list` and reads its pointer to the next element, some parallel subcomputation in `update_list` might be writing to that same pointer to insert an element. This determinacy race means that the

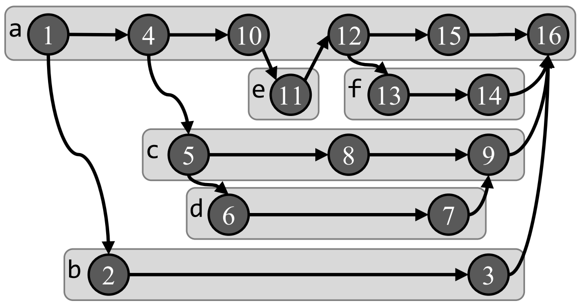


Figure 9-2: Example Cilk computation dag. Dark circles represent strands, and edges represent parallel control dependencies between strands. The strands are numbered according to their serial execution order. Light rectangles correspond to function instantiation and enclose the strands that execute within that instantiation.

`scan_list` can scan a nondeterministic number of elements in the list.

The location where determinacy race occurs is subtle. Because `update_list` employs a reducer to coordinate parallel inserts on its input `list`, any insert into `list` can occur on a distinct view local to the subcomputation performing the insertion. The operation that eventually writes to the pointer at the last element in `list` and realizes the determinacy race is a REDUCE operation that eventually appends to the original view of `list_reducer`, as initialized in line 3. A tool such as Cilk Screen [197] will not catch this particular race, because the determinacy race involves a view-aware instruction executed in a REDUCE operation.

9.3 The PEER-SET algorithm

This section presents the PEER-SET algorithm for detecting view-read races. View-read races are defined formally in terms of the "peer-set semantics" that reducers obey. We describe peer-set semantics in terms of the dag computation model presented in Chapter 2. We formally define the view-read races based on peer-set semantics, and we describe how the PEER-SET algorithm detects such race.

Let us first define some terms. For any two strands u and v in a Cilk computation, we say that u ***precedes*** v, denoted as $u \prec v$, if there exists a path from u to v in the dag. Two strands u and v are logically in ***series*** if either $u \prec v$ or $v \prec u$; otherwise they are logically ***parallel***, denoted as $u \parallel v$. In the example computation dag in Figure 9-2, for example, strands 4 and 9 are logically in series, because strand 4 precedes strand 9, while strands 9 and 10 are logically in parallel. For convenience, we shall assume that strands respect boundaries of ***Cilk functions*** — functions that can spawn — meaning that calling or spawning a Cilk function terminates a strand, as does returning from a Cilk function. Each strand thus belongs to exactly one Cilk function invocation. We shall not worry for now about modeling the execution of view-aware strands in the computation dag.

Peer-set semantics

"Peer-set semantics" dictate which updates are guaranteed to be reflected in the view of a reducer h observed at strand u in terms of the ***peers*** of u — the set of strands in parallel with u, denoted by $peers(u) = \{w \in V : w \parallel u\}$. Conceptually, "peer-set semantics" dictate that the view visible to a strand v is guaranteed to reflect the updates since a previous strand u if u and v have the same peers. In Figure 9-2, for example, these semantics dictate that the view of a reducer at strand 9 is guaranteed to reflect the updates since strand 5, because strands 5 and 9 have the same peers. The view at strand 14, meanwhile, is not guaranteed to reflect the updates since strand 10, because strands 10 and 14 do not share the same peers — strands 12 and 13 are in the peer set of strand 14, but not that of 10. Formally, ***peer-set semantics*** are defined as follows:

Definition 5 (Peer-set semantics) *Let h be a reducer with an associative operator $\otimes$. Consider a serial walk of G, and let $a_1, a_2, \ldots, a_k$ denote the updates to h after the start of instruction u and before the start of instruction v. Let $h(u)$ and $h(v)$ denote the views of h at strands u and v respectively. If $\mathit{peers}(u) = \mathit{peers}(v)$, then $h(v) = h(u) \otimes a_1 \otimes a_2 \otimes \ldots \otimes a_k$.*

One can show, by induction on the updates associated with each view, that the implementation of reducers in Cilk [144] ensures that reducers obey peer-set semantics.

View-read races

Formally, a ***view-read race*** occurs when two accesses to reducers, called "reducer-reads," occur at strands with different peers. Here, we broadly define a ***reducer-read*** as creating a reducer, resetting a reducer's value, or querying the reducer to retrieve its value. On the other hand, invoking Create-Identity, Update, or Reduce on a reducer does not count as a reducer-read, because those functions operate on a reducer's underlying view instead of on the reducer itself. A common instance of a view-read race occurs when between a strand u that queries the value of a reducer and the last strand v before u in the serial execution order to create or reset the reducer's value. In terms of peer-set semantics, a view-read race exists between these two strands if $\mathit{peers}(u) \neq \mathit{peers}(v)$.

Given this definition of a view-read race, a Cilk program with a view-read race might nevertheless behave deterministically. For instance, in the code example shown in Figure 9-1, suppose that the programmer moves the call to `list_reducer.set_value(list)` to after `cilk_spawn` at line 4, thereby creating a view-read race. If `foo` does not modify `list`, however, then the `update_list` routine could behave deterministically, rendering the view-read race ***benign***. We nevertheless declare this to be a race because the reducer-reads violate their peer-set semantics.

Detecting view-read races

The Peer-Set algorithm executes a Cilk computation serially and evaluates its strands in their serial execution order to check for view-read races. The Peer-Set algorithm employs several data structures to track which strands perform reducer-reads and which strands have the same peer set.

During program execution, the Peer-Set algorithm assigns a unique ID to every Cilk function instantiation, and for each instantiated Cilk function F, it maintains a ***shadow frame*** for F on a ***shadow stack***, which is pushed and popped in synchrony with the function-call stack. The shadow frame for F holds two scalars, $F.\mathit{ls}$ and $F.\mathit{as}$, and three "bags," $F.I$, $F.C$, and $F.O$. Each ***bag*** stores a set of ID's for completed function instantiations in a fast disjoint-set data structure [100, Ch. 21]. The contents of these scalars and bags are defined as follows.

- The scalar $F.\mathit{as}$ stores the ***ancestor-spawn count*** — the total number of spawns that each ancestor F' of F has performed since F' last synced.

```
When F calls or spawns G:
1   if $F$ spawns $G$
2       $F.ls$ += 1
3       $F.O \cup= F.C$
4       $F.C = \emptyset$
5   $G.as = F.as + F.ls$
6   $G.ls = 0$
7   $G.I = \textsc{MakeBag}(\{G\})$
8   $G.C = \textsc{MakeBag}(\emptyset)$
9   $G.O = \textsc{MakeBag}(\emptyset)$

When G returns to F:
10  $F.O \cup= G.O$
11  if $F$ spawned $G$
12      $F.O \cup= G.I$
13  elseif $F.ls = 0$
14      $F.I \cup= G.I$
15  else
16      $F.C \cup= G.I$

When F syncs:
17  $F.ls = 0$
18  $F.O \cup= F.C$
19  $F.C = \textsc{MakeBag}(\emptyset)$

When F reads reducer h:
20  if $\textsc{FindBag}(reader(h))$ is a O-bag or
            $reader(h).s \neq F.as + F.ls$
21      a view-read race exists
22  $reader(h) = F$
23  $reader(h).s = F.as + F.ls$
```

Figure 9-3: Pseudocode for the PEER-SET algorithm. The MAKEBAG routine creates a new bag with a specified initial contents. The FINDBAG routine finds the bag containing the specified element by finding the corresponding set in the disjoint-set data structure.

- The scalar $F.ls$ stores the ***local-spawn count*** — the number of spawns F has executed since F last synced.
- The ***I-bag*** $F.I$ contains the ID's of all completed descendants of F with the same peers as the first (i.e., *i*nitial) strand of F.
- The ***C-bag*** $F.C$ contains the ID's of all completed descendants of F with the same peers as the last *c*ontinuation strand executed in F. If F has not spawned since it last synced, then $F.C$ is empty.
- The ***O-bag*** $F.O$ contains the ID's of all *o*ther completed descendants of F not in $F.I$ or $F.C$.

For each Cilk function F, we refer to the sum of the ancestor-spawn and local-spawn counts, $F.as + F.ls$, as the ***spawn count*** of F, which corresponds to the total number of spawn statements executed by F and F's ancestors since each last synced. As an example of ancestor- and local-spawn counts, consider the dag in Figure 9-2. In this dag, the ancestor-spawn count for function c when strand 8 executes is 2, because its ancestor, function a, spawned at strands 1 and 4 before strand 8 executes. The local-spawn count for c, meanwhile, is 1, because strand 5 in c spawned before strand 8 executes. As a result, the spawn count for strand 8 is 3.

The PEER-SET algorithm also maintains a ***shadow space*** of shared memory, called *reader*, which maps each reducer to the strand that accessed it last, along with a corresponding spawn count. Specifically, for each reducer h, $reader(h)$ stores the ID of the Cilk function F that last read h, and the associated field $reader(h).s$ stores the spawn count of F when it last read h.

Figure 9-3 gives the pseudocode of the PEER-SET algorithm, which maintains the bags and scalars for each function frame F as follows. When created, frame F inherits its ancestor-spawn count from the spawn count of its parent, and it initializes its local-spawn count $F.ls$ to 0. As F executes, it increments $F.ls$ when F spawns, and resets $F.l$ to 0 when F syncs. Frame F's bags are updated when a child frame G returns to F, based on whether F has spawned since it last synced. Although the bag $G.O$ is always combined with $F.O$, the bag $G.SS$ is combined with $F.I$ only if F has not spawned since it last synced; otherwise $G.I$ is combined with $F.C$. The bag $G.C$ is guaranteed to be empty when G returns to F because Cilk functions implicitly sync before they return.

The following theorem justifies that the PEER-SET algorithm runs in nearly linear time.

Theorem 61 *Consider a Cilk program that executes in time T on one processor and references x reducer variables. The* PEER-SET *algorithm checks this program execution for a view-read race in $O(T\alpha(x, x))$ time, where α is Tarjan's functional inverse of Ackermann's function.*

PROOF. The pseudocode in Figure 9-3 shows that, at each point in the program execution, the PEER-SET algorithm performs at most a constant number of operations on bags plus a constant amount of additional work. Furthermore, the shadow space *reader* maintained by the PEER-SET algorithm stores only x entries, one for each reducer variable. The theorem follows from a similar analysis as that for the SP-bags algorithm [134, Thm. 1]. □

9.4 Correctness of the Peer-Set algorithm

This section presents a proof that the PEER-SET algorithm correctly detects view-read races. We sketch the intuition for why the PEER-SET algorithm is correct. To formally argue that the PEER-SET is correct, we describe the representation of a Cilk computation dag as an "SP parse tree," and we show how peer-set semantics can be modeled simply within an "SP parse tree." Finally, we argue mathematically for its correctness.

Intuition

To understand how the PEER-SET algorithm works, consider its behavior as it executes a Cilk function. We shall use the dag illustrated in Figure 9-2 as a running example. We shall examine in particular how bags are maintained.

Consider the contents of the bags associated with a Cilk function G when it returns. Because G is guaranteed to be synced, the pseudocode in Figure 9-3 shows that the bag $G.C$ is empty, the bag $G.I$ contains descendants of G with the same peer set as the first strand in G, and the bag $G.O$ contains all other descendants of G. In the dag in Figure 9-2, for example, when `c` returns, bag `c`$.I$ contains the ID for `c`, and bag `c`$.O$ contains the ID for `d`.

When G returns to its caller or spawner F, the PEER-SET algorithm merges the content of $G.O$ and $G.I$ into the bags in its parent F. Let us see how this merger captures the peer-set relationships of these descendant functions with the strands in F.

Because the functions identified in $G.O$ do not share the same peer set as the first strand of G or the last continuation strand executed by G, they must have a different peers from any strand in F, regardless of whether G is called or spawned. Therefore, $G.O$ is always unioned with $F.O$. In the example dag in Figure 9-2, when `c` returns to `a`, unioning bag `c`$.O$ with bag `a`$.O$ correctly identifies that `d` has a distinct peer set from every strand in `a`.

As for $G.I$, we consider two cases.

Suppose that F spawned G. By definition of a spawn, all descendants of G must therefore have a different peer set from strands in F; in particular, the continuation strand in F after the spawn is a peer of the descendants of G. The bag $G.I$ is thus unioned with the bag $F.O$ when G returns. In the example dag in Figure 9-2, because `a` spawned `c` at strand 4,

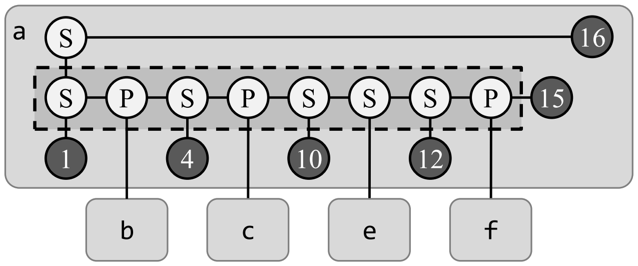


Figure 9-4: The canonical SP parse tree for the function instantiation `a` in the computation dag in Figure 9-2. The internal nodes of a sync block are indicated by the darkened rectangle outlined by a dashed line.

every strand in `c` is in parallel with strand 10, implying that `c` has a distinct peer set from all strands in `a`.

Now suppose that F called G. If F had no outstanding spawned children, then the first strand in G has the same peer set as the first strand in F, and the bag $G.I$ is therefore unioned with $F.I$. Otherwise, F called G when F's local-spawn count was nonzero, meaning that F had at least one outstanding spawned child. The first strand in G therefore has a distinct peer set from that of the first strand in F, but the same peer set as the last continuation strand executed in F. The $G.I$ bag is therefore unioned with $F.C$, where it remains until F either spawns again or syncs. In the example dag in Figure 9-2, strand 11 has a distinct peer set from strand 1, but the same peer set as strand 10, the caller of `e`. When `e` returns to `a`, therefore, unioning the bag `e`.I with `a`.C correctly identifies that the peer set of strand 11 matches that of strand 10.

Ideally we'd like to keep only two bag $F.I$ and $F.O$, for descendants that have the same peers as F's first strand and that don't, but that is insufficient. Consider an example where F spawned off G; before F syncs, it reads a reducer, calls some Cilk function, and reads the reducer again. These two reads share the same peers, but this peer set differs from that of F's first strand. Thus, the PEER-SET algorithms keeps $F.C$ as a special place holder that holds descendants with the same peer set as F's last-executed continuation strand.

Now let us consider detecting a view-read race. If a strand v reads a reducer h, and $\mathit{reader}(h)$ is in some ancestor's O-bag $F.O$, then it certainly has a different peer set from v, and the PEER-SET algorithm thus correctly declares a view-read race. If $\mathit{reader}(h)$ is in $F.I$ for some ancestor F, however, then $\mathit{reader}(h)$ has the same peers as a strand u that is F's first strand, and the currently executing strand v may or may not have the same peers as u. To handle this case, the PEER-SET algorithm compares the spawn count of v against the spawn count of $\mathit{reader}(h)$ stored in $\mathit{reader}(h).s$, which must match the spawn count as u. As long as v has this same spawn count, then no ancestor of v below F added a peer to v that is not a peer of u, meaning that u and v have the same peer set. A similar argument holds for $F.C$ and the strand u that is the last continuation strand executed in F.

SP parse trees

To argue formally for the PEER-SET algorithm's correctness, we adopt the representation of a Cilk computation dag as an "SP parse tree" as introduced by Feng and Leiserson [134]. As Feng and Leiserson show, the dag modeling a Cilk computation (that does not use reducers) is a ***series-parallel dag***, which has a distinguished ***source*** vertex s and a distinguished ***sink*** vertex t and can be constructed recursively with series and parallel compositions. This recursive construction can be represented by a binary tree, which we call an ***SP parse tree***.

Figure 9-4 illustrates the SP parse tree corresponding to function a in the dag in Figure 9-2. The leaves of the SP parse tree are strands in the dag, and each internal node is either an S-node, denoting a series composition of its two children, or a P-node, denoting parallel composition of its two children. The SP parse tree in Figure 9-4 is a ***canonical*** parse tree [134], meaning that its internal nodes are laid out as follows. The sync strands in a Cilk function F partition the strands in F into ***sync blocks***. The canonical SP parse subtree for a sync block is a chain of S-nodes and P-nodes, where the left child of each node is either a strand in F or the root of the canonical parse tree for a subcomputation spawned or called in F, and the right child is the next S- or P-node at the root of the SP parse subtree for the vertices following the left subchild in the serial order. A chain of S-nodes, called the ***spine***, links the sync blocks within F.

Feng and Leiserson prove the following lemma [134, Lemma 4] that shows that the series-parallel relationship between two strands u and v in a Cilk computation is encoded in their ***least common ancestor*** in the SP parse tree, denoted $\mathrm{LCA}(u, v)$, which is the deepest node in the tree that is a common ancestor of both u and v.

Lemma 62 *Let u and v be distinct strands in a Cilk computation dag, and let $\mathrm{LCA}(u, v)$ be their least common ancestor in the SP parse tree for the dag. Then $u \parallel v$ if and only if $\mathrm{LCA}(u, v)$ is a P-node.*

Proof of correctness

Peer-set semantics can be modeled simply in terms of the SP-parse-tree representation of a Cilk computation. The following lemma relates peer-sets to the SP parse tree. In particular, we show that two strands share the same peer set if and only if the path connecting them in the SP parse tree consists only of S-nodes.

Lemma 63 *Two strands u and v have the same peer set, $peers(u) = peers(v)$, if and only if the path connecting u to v in the SP parse tree consists entirely of S-nodes.*

PROOF. We first argue that $\mathrm{LCA}(u, v)$ must be an S-node. If $\mathrm{LCA}(u, v)$ is a P-node, then $u \parallel v$, and therefore $u \in peers(v)$. Because $u \notin peers(u)$, we have that $peers(u) \neq peers(v)$.

($\Rightarrow$) Suppose that the path in the SP parse tree from $\mathrm{LCA}(u, v)$ to u contains a P-node. Then there must exist a strand w such that $\mathrm{LCA}(u, w)$ is this P-node, which implies that $u \parallel w$ and, therefore, that $w \in peers(u)$. Because this P-node is on the path from $\mathrm{LCA}(u, v)$ to u, we have that $\mathrm{LCA}(w, v) = \mathrm{LCA}(u, v)$, which is an S-node. Therefore, $w \not\parallel v$, and thus $w \notin peers(v)$. This P-node therefore implies that $peers(u) \neq peers(v)$, so if $peers(u) = peers(v)$, then no such P-node can exist. A symmetric argument shows that no P-node can exist on the path from $\mathrm{LCA}(u, v)$ to v.

($\Leftarrow$) Suppose that $peers(u) \neq peers(v)$, and without loss of generality, suppose that u executes before v in the serial order. If $v \in peers(u)$, then $u \parallel v$ and $\mathrm{LCA}(u, v)$ is a P-node. Otherwise, we have $u \prec v$ and there exists some strand w in exactly one of $peers(u)$ or $peers(v)$. Suppose that $w \in peers(u)$ and $w \notin peers(v)$. Then $w \parallel u$ and $w \not\parallel v$. By Lemma 62, we have that $\mathrm{LCA}(w, u)$ is a P-node and $\mathrm{LCA}(w, v)$ is an S-node. The nodes $\mathrm{LCA}(w, u)$ and $\mathrm{LCA}(w, v)$ therefore differ, and one can show that $\mathrm{LCA}(u, v)$ is one of these two. Either way, the P-node $\mathrm{LCA}(w, u)$ appears on the path from u to v in the SP parse tree. The case where $w \notin peers(u)$ and $w \in peers(v)$ is similar. $\square$

We now argue that the PEER-SET algorithm identifies strands that are connected by S-nodes in the SP parse tree, which implies that they share the same peer set. As in [134], we define the ***procedurification*** function $\mathcal{F}$ as the map from strands and nodes in the SP parse tree to Cilk function instantiations.

Lemma 64 *Consider an execution of the* PEER-SET *algorithm on a Cilk computation. Suppose that strand u executes before strand v, and let $\mathcal{F}$ be the procedurification function mapping the SP parse tree to Cilk function invocations. Let $a = \mathrm{LCA}(u, v)$ be the least common ancestor of u and v in the SP parse tree. Then both of the following conditions hold if and only if the path from u to v in the SP parse tree consists entirely of S nodes.*

1. *The ID for $\mathcal{F}(u)$ belongs to either the I-bag or the C-bag of $\mathcal{F}(a)$ when v executes.*
2. *The spawn count for $F(a)$ when u executes equals the spawn count for $\mathcal{F}(v)$ when v executes.*

PROOF. ($\Rightarrow$) We first show that, if there exists a P-node on the path from u to v in the SP parse tree, then one of the conditions is violated. Let b be the first such P-node.

Suppose that b lies on the path between a (inclusive) and u. Then u must lie in a subtree of b. If u is in the left subtree of b, then $\mathcal{F}(b)$ spawned a function F' which is either $\mathcal{F}(u)$ or an ancestor of $\mathcal{F}(u)$. The pseudocode in Figure 9-3 shows that, when F' returns, the procedure ID of $\mathcal{F}(u)$ is placed in the O-bag of $\mathcal{F}(b)$, violating condition 1. Suppose instead that u is in the right subtree of b. Then the PEER-SET algorithm's pseudocode in Figure 9-3 shows that $\mathcal{F}(u)$ is added to either a O- or an C-bag. Moreover, v is not in a subtree of b, because u executes before v in the serial order. Because b is a P-node, the structure of the canonical SP parse tree implies that b lies inside a sync block and all strands in the right subtree of b are in or invoked from the same sync block. Therefore, u and v are in subtrees under different sync blocks in $\mathcal{F}(b)$, which implies that the PEER-SET algorithm must have executed a sync between the time it executed u and the time it executed v. From the pseudocode in Figure 9-3, $\mathcal{F}(u)$ is placed in a O-bag when this sync executes. Because no action of the pseudocode moves ID's from a O-bag into either an I- or C-bag, $\mathcal{F}(u)$ always lies in the O-bag of an ancestor of $\mathcal{F}(b)$ after $\mathcal{F}(b)$ returns, violating condition 1.

Suppose instead that b lies on the path between a (exclusive) and v. Then b must lie in the right subtree of a by construction of the canonical SP parse tree. Consequently, $\mathcal{F}(b)$ spawned a child Cilk function invocation F' that it did not sync before v executed. From the pseudocode in Figure 9-3, $\mathcal{F}(b)$ incremented its local-spawn count when it spawned F'. The spawn count for $\mathcal{F}(v)$ is therefore at least one larger than that of $\mathcal{F}(a)$ when u executed, violating condition 2.

($\Leftarrow$) We now show that, if one of the conditions is violated, then there exists a P-node on the path in the SP parse tree from u to v.

Suppose that the spawn count for $\mathcal{F}(a)$ when u executes does not match that for $\mathcal{F}(v)$ when v executes, violating condition 2. Then either $F'.ls \neq 0$ for some ancestor F' of $\mathcal{F}(v)$ below and including $\mathcal{F}(a)$, or $\mathcal{F}(a).ls$ changed value between executing u and v. In the first case, F' spawned a child computation that it did not sync before v executed. By construction of the SP parse tree, there therefore exists a P node on the path from a to v. In the second case, if $\mathcal{F}(a).ls$ increased, then before v executed, $\mathcal{F}(a)$ spawned a subcomputation in the same sync block that invoked the subcomputation containing v, and a P node therefore exists on the path from a to v. Otherwise, $\mathcal{F}(a).ls$ was non-zero when u executed and was reset to 0 between the executions of u and v, implying that $\mathcal{F}(a)$ executed a sync between executing u and v. By construction of the canonical SP parse tree, a P node therefore lies on the path from a to u.

Now suppose that the ID for $\mathcal{F}(u)$ belongs to a O-bag of $\mathcal{F}(a)$ when v executes, violating condition 1. If $\mathcal{F}(u)$ is in a O-bag, then the pseudocode in Figure 9-3 shows that $\mathcal{F}(u)$ must have been placed in a O-bag or an C-bag of one of its ancestors F' below and including $\mathcal{F}(a)$. Consequently, the pseudocode implies that $F'.\mathit{ls}$ was nonzero when $\mathcal{F}(u)$ was added to one of its bags, meaning that F' spawned a subcomputation that it did not sync before u executed. If F' is below $\mathcal{F}(a)$, then there must therefore be a P-node on the path from u to a. Otherwise $F' = \mathcal{F}(a)$, and either $\mathcal{F}(a)$ spawned the subcomputation containing u, implying that a is a P-node, or $\mathcal{F}(a).\mathit{ls}$ changed value between the executions of u and v, in which case the argument above shows that a P-node lies on the path from u to v. $\square$

We now argue that the PEER-SET algorithm detects a view-read race if only if one exists.

Theorem 65 *The* PEER-SET *algorithm detects a view-read race in a Cilk computation if and only if a view-read race exists.*

PROOF. Let $\mathcal{F}$ be the procedurification function mapping the SP parse tree to Cilk function invocations.

($\Rightarrow$) We first argue that, if the PEER-SET algorithm detects a view-read race, then one exists. If the PEER-SET algorithm detects a view-read race on a reducer h when executing strand v, then Figure 9-3 shows that either *reader* (h) belongs to a O-bag or the spawn count *reader* $(h)\,.\,s$ does not match $\mathcal{F}(v).\mathit{as} + \mathcal{F}(v).\mathit{ls}$. Lemma 64 therefore implies that a P node exists on the path from u to v in the SP parse tree, meaning that $\mathit{peers}(u) \neq \mathit{peers}(v)$ by Lemma 63. Consequently, a view-read race exists.

($\Leftarrow$) We now argue that, if a view-read race exists on a reducer h, then the PEER-SET algorithm detects it. Let u and v be two strands involved in a view-read race on reducer h, where u executes before v in the serial order and, if several such races exist, we choose the race for which v executes earliest in the serial order. The definition of a view-read race implies that $\mathit{peers}(u) \neq \mathit{peers}(v)$.

When v executes, suppose that *reader* $(h) = \mathcal{F}(w)$ for some strand w. If $w = u$, then because $\mathit{peers}(u) \neq \mathit{peers}(v)$, Lemmas 63 and 64 imply that a view-read race is reported. If $w \neq u$, then w must have executed after u in the serial order, in order to overwrite *reader* (h). We must also have that $\mathit{peers}(w) = \mathit{peers}(u)$; otherwise Lemmas 63 and 64 imply that a view-read race exists between w and u, and the fact that both w and u executed before v contradicts strand v being the earliest strand in the serial order for which a view-read race exists on h. Because $\mathit{peers}(u) \neq \mathit{peers}(v)$, we have that $\mathit{peers}(w) \neq \mathit{peers}(v)$, and by Lemmas 63 and 64, a view-read race is detected. $\square$

9.5 The SP+ algorithm

This section presents the SP+ algorithm for detecting determinacy races in Cilk computations that use reducers. A parallel execution of a Cilk program that uses reducers contains view-aware strands, both from calls to update the reducer in the user code and from runtime-invoked CREATE-IDENTITY and REDUCE operations. The SP+ algorithm extends the SP-bags algorithm [134] to handle this additional complexity. The SP+ algorithm models the execution of a Cilk program that uses a reducer as a “performance dag,” which was introduced in Chapter 3. We review the salient details of this performance-dag model here.We identify the circumstances in which a determinacy race can exist in a computation that uses reducers. We describe how the SP+ algorithm detects such determinacy races. Finally, we provide some high-level intuition for the SP+ algorithm’s correctness.

To handle the nondeterminism of how the Cilk runtime system manages views of a reducer, the SP+ algorithm takes a steal specification as part of its input. The steal specification dictates which continuations in the computation are stolen, when new reducer views are created, and the partial order in which views are reduced together. The steal specification thereby identifies a particular execution of a Cilk program that uses a reducer. We shall refer to the particular execution of a Cilk program dictated by a steal specification as a ***steal-specified Cilk computation***.

To simplify the description, we shall assume that the Cilk program uses a single reducer, and that the REDUCE, CREATE-IDENTITY, and UPDATE methods all execute serial code.[3] As a result, the execution of each of these functions can be modeled as a single strand in the computation dag. As in Chapter 3, we shall refer to a strand that arises from the runtime system's implicit execution of REDUCE as a ***reduce strand***, and a strand that arises from the runtime's implicit execution of CREATE-IDENTITY as an ***init strand***.

Review the performance dag model

To describe the SP+ algorithm, we build upon the execution model for Cilk programs that use reducers introduced in Chapter 3, which models a computation as a "performance dag." The "performance dag" model first represents the Cilk computation A as an ordinary computation dag called the ***user dag*** $A_\nu = (V_\nu, E_\nu)$, which does not contain any reduce or init strands. Let $h(v)$ denotes the view of h that a strand $v \in V_\nu$ can access. In terms of the user dag, the runtime system tracks the views of a reducer h according to the following invariants, which Chapter 3 presented and we reiterate here:

Invariant 1 *If $u \in V_\nu$ has out-degree 1 and $(u, v) \in E_\nu$, then $h(v) = h(u)$.*

Invariant 2 *Suppose that $u \in V_\nu$ is a spawn strand with outgoing edges $(u, v), (u, w) \in E_\nu$, where $v \in V_\nu$ is the first strand of the spawned subroutine and $w \in V_\nu$ is the continuation in the parent. Then, we have $h(v) = h(u)$ and*

$$h(w) = \begin{cases} h(u) & \textit{if } u \textit{ was not stolen} \\ \textit{new view} & \textit{otherwise.} \end{cases}$$

Invariant 3 *If $v \in V_\nu$ is a sync strand, then $h(v) = h(u)$, where u is the first strand of v's function.*

When a new view $h(w)$ is created according to Invariant 2, the new view $h(w)$ is a ***parallel view*** to $h(u)$. We say that the old view $h(u)$ ***dominates*** $h(w)$, which we denote by $h(u) > h(w)$. For a set H of views, we say that two views $h_1, h_2 \in H$ are ***adjacent*** if there does not exist $h_3 \in H$ such that $h_1 > h_3 > h_2$.

For each parallel view created according to Invariant 2, the runtime system eventually calls REDUCE to reduce that view with another view. Let u be a spawn strand with outgoing edges $(u, v), (u, w) \in E_\nu$, where $v \in V_\nu$ is the first strand of the spawned subroutine and $w \in V_\nu$ is the continuation in the parent. Let y be the sync strand to which the subroutine spawned by u returns. If $h(w)$ is a new view, then the runtime system executes a REDUCE operation to reduce $h(w)$ before executing y. As Chapter 3 describes, each reduce strand reduces two adjacent views h_1 and h_2 together, destroying the dominated view in the pair.

[3]It is generally the case in practice that the REDUCE and CREATE-IDENTITY methods of a reducer execute serial code.

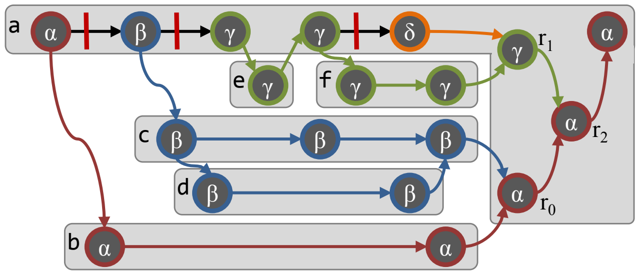


Figure 9-5: An example of performance dag, which corresponds to augmenting the user dag in Figure 9-2 in Section 9.2 with reduce strands r_0, r_1, and r_2, and init strands (not shown). A vertical bar across an edge indicates that the following continuation strand is stolen. Each strand is labeled with its associated view ID. Strands with the same view ID are highlighted with the same color.

Chapter 3 describes precisely how the user dag $A_\nu = (V_\nu, E_\nu)$ of a steal-specified Cilk computation A can be augmented into a ***performance dag*** $A_\pi = (V_\pi, E_\pi)$ that additionally models reduce and init strands. The primary complication in the performance dag concerns the modeling of reduce strands. For each sync strand v in A_ν, a set R of reduce strands must execute before v to destroy the views of the views of the immediate predecessors of v in A_ν. For each reduce strand $r \in R$, we say that v is the sync ***responsible*** for r. Chapter 3 describes how the reduce strands in R form a ***reduce tree***, a rooted binary tree interposed in the performance dag between v and the immediate predecessors of v in A_ν. In particular, each predecessor of v in the user dag A_ν becomes a predecessor of a leaf of the reduce tree in A_π, and the root of the reduce tree becomes the immediate predecessor of v.

Figure 9-5 illustrates an example of a performance dag, which corresponds to augmenting the user dag shown in Figure 9-2 with reduce strands. In this dag, three different continuation points in function a are stolen, each causing a new view to be generated, leading to a total of 4 views in a. For each newly created view, a corresponding reduce strand in a destroys that view. The reduce strand r0, for example, reduces the views α and β, destroying β and inheriting the view ID α. Figure 9-5 also shows how these reduce strands form a reduce tree before the final strand in a, which is a sync strand. The reduce strands and the structure of the reduce trees are both functions of the execution schedule, which is fixed by the input steal specification.

Determinacy races that involve view-aware strands

View-aware strands complicate the circumstances under which a determinacy race occurs. For example, in the performance dag shown in Figure 9-5, let u be the first strand in function d, and let v be the second strand in function c. Suppose that u and v access the same memory location ℓ with one being a write, and suppose that v is a view-aware strand generated from executing UPDATE. Because v is a continuation that is not stolen in this execution, the same worker executes v immediately after returning from d, and both u and v observe the same view β of the reducer, as Figure 9-5 shows. Because v is view-aware, in a different execution in which v is stolen, v would observe a different view and might therefore write to different memory locations. If location ℓ is part of view β, for example, then in this alternate scenario, v might not write to ℓ, precluding a determinacy race with u. Because v is view-aware, its logical parallelism with u is not sufficient for it to definitively race with u; it must also operate on a parallel view.

We summarize the conditions under which a determinacy race exists between two strands u and v in a Cilk computation that uses a reducer. Suppose that v follows u in the serial

```
F spawns or calls G:
24  $G.S = \textsc{MakeBag}(\{G\}, \textsc{Top}(F).vid)$
25  $p = \textsc{MakeBag}(\emptyset, \textsc{Top}(F).vid)$
26  $G.P = \langle p \rangle$

F syncs:
27  $F.S \cup= \textsc{Top}(F.P)$
28  $p = \textsc{MakeBag}(\emptyset, F.S.vid)$
29  $\textsc{Top}(F.P) = p$

Spawned G returns to F:
30  $\textsc{Top}(F.P) \cup= G.S$

Called G returns to F:
31  $F.S \cup= G.S$

F executes a stolen continuation:
32  $p = \textsc{MakeBag}(\emptyset, \textbf{new}$ view ID$)$
33  $\textsc{Push}(F.P, p)$

Reduce R returns to F:
37  $\textsc{Top}(F.P) \cup= R.S$

F calls Reduce R:
34  $p = \textsc{Pop}(F.P)$
35  $\textsc{Top}(F.P) \cup= p$
36  $R.S = \textsc{MakeBag}(\{R\}, \textsc{Top}(F.P).vid)$
```

Figure 9-6: Pseudocode for the SP+ algorithm to maintain bags. Each bag is a set with a *vid* field, which tracks the view ID of that bag. (The view ID of an S-bag matches that of the first P-bag in the P-stack.) This *vid* field is set when the bag is first created and remains invariant as the bag's contents are modified. In particular, when two P-bags are unioned together, the view ID of the destination P-bag is preserved. For a given P-stack x, PUSH(x) pushes an element on top of x, and POP(x) pops x. TOP(x) reads the topmost bag of x without modifying x. MAKEBAG(S, v) creates a new bag with view ID v that contains the elements of S.

order, both u and v access the same location ℓ, and at least one of them writes to ℓ.

- If v is a view-oblivious strand, then a determinacy race exists between u and v if and only if u and v are logically in parallel.
- If v is a view-aware strand, then a determinacy race exists between u and v if and only if u and v are both logically in parallel and $h(u)$ and $h(v)$ are distinct, parallel views of reducer h.

Detecting determinacy races that involve reducers

Like the PEER-SET and SP-bags algorithms, the SP+ algorithm is a serial algorithm that evaluates the strands of a Cilk computation in their serial order to detect determinacy races. As it executes, SP+ employs several data structures to keep track of the parallel views created in a steal-specified Cilk computation and to determine the series-parallel relationships between strands, including reduce strands.

Like the SP-bags algorithm, SP+ maintains two shadow spaces of shared memory, called *reader* and *writer*. Each shadow space contains an entry for each memory location that the computation accesses. During the execution, each Cilk function instantiation is given a unique ID. Each location ℓ in *reader* stores the ID of the function instantiation that last read ℓ, while each location ℓ in *writer* stores the ID for the function instantiation that last wrote ℓ.

For each Cilk function F on the call stack, the SP+ algorithm also maintains a shadow frame containing a set of bags. Each bag stores a set of ID's for completed procedures in a fast disjoint-set data structure [100, Ch. 21]. In particular, when executing a strand u, the bags associated with a function F on the call stack have the following contents:

- The ***S-bag*** $F.S$ contains the ID's of F's completed descendants that precede u, as well as the ID for F itself.
- The ***P-stack*** $F.P$ contains a stack of ***P-bags***. Together, the P-bags in $F.P$ contain the set of ID's of F's completed descendants that are logically in parallel with u. The separate P-bags $p \in F.P$ partition this set into subsets based on the parallel views

```
read a shared location $\ell$ by a view-oblivious strand in $F$:
38 if FindBag($writer(\ell)$) is a P-bag
39     a determinacy race exists
40 if FindBag($reader(\ell)$) is an S-bag
41     $reader(\ell) = F$

write a shared location $\ell$ by a view-oblivious strand in $F$:
42 if FindBag($reader(\ell)$) is a P-bag or FindBag($writer(\ell)$) is a P-bag
43     a determinacy race exists
44 if FindBag($writer(\ell)$) is an S-bag
45     $writer(\ell) = F$

read a shared location $\ell$ by a view-aware strand in $F$:
46 if FindBag($writer(\ell)$) is a P-bag and FindBag($writer(\ell)$).$vid \neq$ Top($F.P$).$vid$
47     a determinacy race exists
48 if FindBag($reader(\ell)$) is an S-bag or
           ($F$ is an invocation of Reduce and FindBag($reader(\ell)$).$vid ==$ Top($F.P$).$vid$)
49     $reader(\ell) = F$

write a shared location $\ell$ by a view-aware strand in $F$:
50 if FindBag($reader(\ell)$) is a P-bag and FindBag($reader(\ell)$).$vid \neq$ Top($F.P$).$vid$
51     a determinacy race exists
52 if FindBag($writer(\ell)$) is a P-bag and FindBag($writer(\ell)$).$vid \neq$ Top($F.P$).$vid$
53     a determinacy race exists
54 if FindBag($writer(\ell)$) is an S-bag or
           ($F$ is an invocation of Reduce and FindBag($writer(\ell)$).$vid ==$ Top($F.P$).$vid$)
55     $writer(\ell) = F$
```

Figure 9-7: Pseudocode for the SP+ algorithm to detect races. As described in the corresponding pseudocode in Figure 9-6, each bag is a set with a *vid* field, which tracks the view ID of that bag. For a P-stack x, Top($F.P$) reads the topmost P-bag of $F.P$ without modifying $F.P$. FindBag(f) finds the bag that contains f.

created.

Figures 9-6 and 9-7 give the pseudocode of the SP+ algorithm, where Figure 9-6 gives the pseudocode for maintaining bags, and Figure 9-7 gives the pseudocode for detecting races. Like the SP-bags algorithm, the SP+ algorithm pushes and pops shadow frames onto a shadow stack in synchrony with the program execution pushing and popping Cilk functions on the call stack.

The SP+ algorithm extends the SP-bags algorithm to additionally push and pop P-bags on a P-stack when reducer views are created or reduced together. Conceptually, each P-stack in SP+ replaces a P-bag in the SP-bags algorithm in order to keep track of views. Each P-bag p has an associated ***view ID***, denoted $p.vid$, which is a unique ID associated with the P-bag on its creation. Executing a stolen continuation pushes a new P-bag with a new view ID onto the top of the P stack. Executing a Reduce operation in F combines the top two P-bags in the P-stack $F.P$, unioning the newer P-bag into the older one.

Determinacy races are detected by the code in Figure 9-7. As the pseudocode shows, different codes are used depending on whether the second strand is view-oblivious or view-aware.

Let us examine how the SP+ algorithm operates by supposing it executes on the computation modeled by the performance dag shown in Figure 9-5. When it executes the fifth strand u in function `a` — the stolen continuation labeled with δ — it pushes a new empty

P-bag corresponding to view δ. At this point, the P-stack $\mathsf{a}.P$ contains two other P-bags: $\{\mathsf{b},\mathsf{c},\mathsf{d}\}$, associated with view α, and $\{\mathsf{e},\mathsf{f}\}$, associated with view γ. The first P-bag resulted from unioning the P-bags corresponding to views α and β before executing r_0. After SP+ executes u and encounters r_1, the steal specification dictates that the top two P-bags — the empty one representing strand u and the one containing $\{\mathsf{e},\mathsf{f}\}$ — are unioned before executing r_1. Thus, if r_1, a view-aware strand, happens to write to location ℓ last accessed by the first strand in f labeled with γ, SP+ will not report a race, because f belongs to the top P-bag of a when r_1 executes. If the last access of ℓ before r_1 is performed by a strand in c, however, a race will be reported, since c is not in the top P-bag of a.

The following theorem analyzes the running time of the SP+ algorithm.

Theorem 66 *For a steal-specified Cilk computation A, let $\mathrm{Work}(A_\pi)$ denote the work of the performance dag of A, and let v denote the number of shared memory locations accessed by A. The SP+ algorithm checks A for a determinacy race in $O(\mathrm{Work}(A_\pi)\cdot\alpha(v,v))$ time.*

Proof. The pseudocode in Figures 9-6 and 9-7 shows that, at each point in the program execution, the Peer-Set algorithm performs at most a constant number of operations on bags plus a constant amount of additional work. The theorem thus follows from a similar analysis as that for the SP-bags algorithm [134, Thm. 1]. □

Corollary 67 *For a steal-specified Cilk computation A, let $T = \mathrm{Work}(A_\nu)$ denote the work of the user dag of A, let M be the number of specified steals, and let τ be the worst-case running time of any* Reduce *or* Create-Identity *operation. The* SP+ *the number of shared memory locations accessed by A. The* SP+ *algorithm checks A for a determinacy race in $O((T+M\tau)\alpha(v,v))$ time.*

Proof. Each specified steal can incur one Create-Identity operation and Reduce operation, which are not accounted for in the user dag. The work of the performance dag is therefore $O(\mathrm{Work}(A_\nu)+M\tau)$, and the corollary thus follows from Theorem 66. □

Intuition for correctness

With respect to detecting races between two view-oblivious strands, it is straightforward to see that SP+ provides the same correctness guarantee as the SP-bags algorithm [134]. Like the SP-bags algorithm, as it executes, SP+ maintains, for every active Cilk function F, two sets of ID's corresponding to F's completed descendants: one set (in the S-bag $F.S$) for those that are logically in series with the currently executing strand, and one set (in the P-stack $F.P$) for those that are logically in parallel with that strand. Both SP-bags and SP+ maintain these sets and use them to detect determinacy races between view-oblivious strands in effectively the same way. SP+ differs only in that it partitions the strands that are logically in parallel across multiple P-bags.

With respect to detecting races between a view-oblivious strand and a view-aware strand, SP+ needs to manage multiple P-bags per Cilk function to handle two complications arising from the use of reducers. First, when a view-aware strand is involved, a race between two strands exists only if two conditions are met: the strands are logically in parallel, and they operate on parallel views. Consequently, the SP+ algorithm must keep track of the views that strands might operate on. Second, the SP+ algorithm must also keep track of different sets of strands within the a Cilk function F that may end up serialized with some reduce strand executed in F.

SP+ maintains P-bags and their concomitant view ID's in a manner that imitates the Cilk runtime's management of views. Each P-bag has a view ID. When a function F is first spawned or called, it inherits the same view ID as its parent's top P-bag. Whenever SP+ executes a stolen continuation in F in the steal-specified Cilk computation, it pushes a new P-bag onto the top of $F.P$ with a brand new view ID. For a currently executing function F, its top P-bag thus has the view ID corresponding to the view of its currently executing strand. Whenever a REDUCE operation occurs in the steal-specified Cilk computation, the SP+ algorithm pops the top P-bag off of $F.P$ and unions it into the next P-bag on top, imitating how a reduce strand combines adjacent views and destroys the dominated view. Because a necessary set of REDUCE operations must occur to destroy all parallel views that reach a sync, when F syncs, SP+ maintains the invariant that only a single P-bag is left in $F.P$, which is the same P-bag (with the same view ID) that F had when it started. The view ID effectively simulates how the runtime manages views.

In addition to keeping track of parallel views via view ID's, the multiple P-bags differentiate the sets of strands that can serialize with different REDUCE operations. Specifically, whenever a spawned function G returns to F, the ID's corresponding to G's descendants, including G itself, get unioned into F's top P-bag. Each P-bag in $F.P$ thus contains a set of ID's corresponding to F's descendants whose initial strands share the same view. Whenever a REDUCE operation occurs, the top two P-bags have the view ID's corresponding to the views that the REDUCE operation will combine, and the set of ID's they contain correspond to the set of F's descendants that serialize with the reduce operation. Because everything that comes after this reduce strand, including this reduce strand, is in series with the descendants corresponding to the ID's in the top two P-bags, SP+ can safely union them together immediately before the reduce strand executes.

To detect a potential race with a view-aware strand, SP+ checks that not only are the two strands in parallel, but also that they operate on parallel views, as verified by comparing the view ID's of the last access and currently executing strand. Note that the union of the top two P-bags occurs *before* the invocation of the corresponding REDUCE operation, and thus any memory access performed by the reduce strand will have the same view ID's as the descendants in those P bags, achieving the desired effect — the reduce strand is in series with descendants in these two P-bags.

Figure 9-7 shows one subtlety in how SP+ handles the shadow memory. SP+ replaces the last reader and writer only if the last access is in an S-bag *or* if the current access is performed by a reduce strand that shares the same view as the last access. Call the last access in the shadow memory u and the currently executing strand v. By "pseudotransitivity of $\|$" [134], we know that there is no need to replace u in the shadow memory with v if v is logically in parallel with u, because any strand that comes later in serial order that races with v will race with u as well. We need only to update the last reader/writer if v is in series with u. In the case where v is a reduce strand, however, v is in series with u even if u belongs to a P-bag, as long as the P-bag shares the same view ID.

9.6 The spawn parse tree and the view parse tree

To argue formally that the SP+ algorithm is correct, we must show that the execution of SP+ captures the series-parallel relationships between strands and their views in the performance dag. This section introduces the "spawn parse tree" and "view parse tree" to capture these series-parallel relationships between strands and views, respectively, in

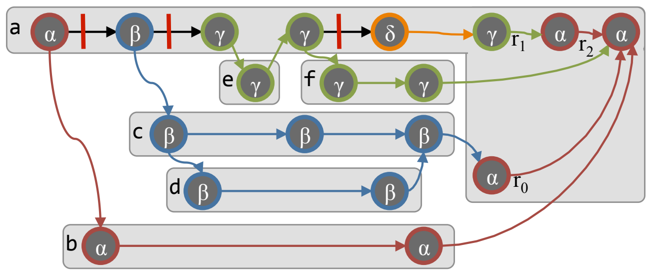


Figure 9-8: The spawn dag for the performance dag in Figure 9-5. Strands are labeled and colored similarly as in Figure 9-5.

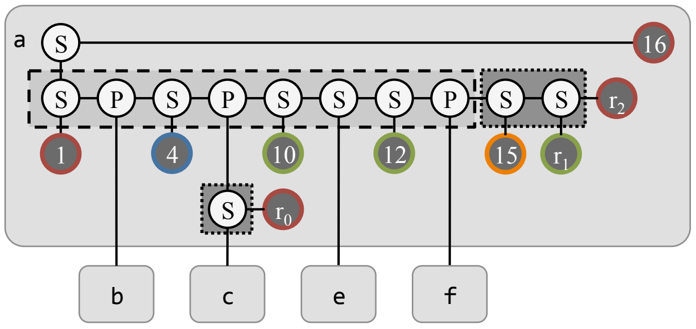


Figure 9-9: The spawn parse tree for the performance dag in Figure 9-5. This spawn parse tree augments the SP parse tree in Figure 9-4 with reduce chains, which are indicated by the dark rectangles outlined by dotted lines. Strands, which appear at the leaves of the tree, are labeled with either their label in Figure 9-2 or, for reduce strands, by their label r_0, r_1, or r_2 in Figure 9-5. Strands with the same view ID are colored similarly, as in Figure 9-5.

a performance dag. We illustrate some example parse trees, and we show several useful properties that these trees exhibit.

The spawn parse tree and view parse tree together address two complications in showing that the SP+ algorithm correctly detects determinacy races in a steal-specified Cilk computation. First, unlike the SP-bags algorithm, the SP+ algorithm must identify when a view-aware strand can modify its view in parallel with another strand, that is, when two strands operate on distinct, parallel views. Second, unlike its user dag, the performance dag is not in general a series-parallel dag, and thus the logical series-parallel relationships between strands cannot be simply represented using an SP parse tree.

Intuitively, the spawn parse tree maintains the series-parallel relationships among strands in a performance dag, and the view parse tree identifies which strands can operate on their views in parallel. Although this intuition breaks down for reduce strands, the spawn and view parse trees together suffice to identify races involving reduce strands.

The spawn parse tree

The ***spawn parse tree*** augments the SP parse tree with reduce strands to model nearly all of the series-parallel relationships between strands in a performance dag. A spawn parse tree correctly models all of the series-parallel relationships between non-reduce strands, as well as some of the relationships involving reduce strands. In particular, the spawn parse tree captures the series-parallel relationships between strands based on `cilk_spawn` statements.

The spawn parse tree of a performance dag A_π is constructed from a version of the performance dag, called the ***spawn dag***, in which some edges have been replaced to produce a series-parallel dag. For example, Figure 9-8 illustrates the spawn dag for the performance dag in Figure 9-5. As Figure 9-8 illustrates, for each reduce strand r, all but one edge into r is replaced with an edge into the sync strand responsible for r. The one edge into r that is not modified is the edge from the last strand to execute before r in the serial order. Formally, for each reduce strand r in the performance dag, let $u_1, u_2, \ldots, u_k$ be the k predecessors of r in the serial order, and let v denote the sync strand responsible for r. For each of the $k-1$ edges (u, r) where $u = u_1, u_2, \ldots, u_{k-1}$, replace edge (u, r) with an edge (u, v).

The following lemma shows that spawn dag is a series-parallel dag.

Lemma 68 *The spawn dag corresponding to a given steal-specified Cilk computation is a series-parallel dag.*

PROOF. To construct a spawn dag recursively using series and parallel compositions, for each function F in the spawn dag, the reduce strands in F before a particular sync strand y are composed in series with F and its spawned subcomputations that sync at y. □

The spawn parse tree of a performance dag is the canonical SP parse tree for the corresponding spawn dag. Figure 9-9 illustrates the spawn parse tree for the performance dag shown in Figure 9-5. Similarly to how a performance dag A_π augments a corresponding user dag A_ν with reduce and init strands, one can view the spawn parse tree as adding reduce and init strands to the SP parse tree of A_ν. This property is illustrated in the comparison between the spawn parse tree in Figure 9-9 and the SP parse tree in Figure 9-4. As Figure 9-9 shows, the spawn parse tree appends chains of S-nodes, called ***reduce chains***, to each sync block in its corresponding SP parse tree. These reduce chains connect the existing strands in the SP parse tree to the reduce strands in a manner that reflects their series-parallel relationships in the spawn dag. Furthermore, because the edge into each reduce strand r in the spawn dag comes from the last predecessor of r in the serial execution order, the serial execution order of the strands in the performance dag and spawn dag are the same. Consequently, a serial execution of the performance dag corresponds to the depth-first, left-to-right traversal of its corresponding spawn parse tree.

The spawn dag captures a subset of the parallel control dependencies between strands in the performance dag, but it can mistakenly show reduce strands as being in parallel with other strands. In the spawn parse tree in Figure 9-9, for example, the least common ancestor between r_1 and any strand in `f` is a P node, even though the performance dag in Figure 9-5 shows that r_1 follows the entire execution of `f`.

The following two lemmas show that, except for reduce strands, two strands are logically in parallel in the performance dag if and only if their least-common ancestor in the spawn parse tree, denoted as $\mathrm{LCA_S}$, is a P-node.

Lemma 69 *Let u and v be strands in a performance dag, where v follows u in the serial execution order. If $u \parallel v$ then $\mathrm{LCA_S}(u, v)$ is a P-node.*

PROOF. The spawn dag is a version of the performance dag where some paths have been removed. Consequently, if $u \parallel v$, then no path connects u to v in either the performance dag or the spawn dag. The construction of the spawn parse tree from the spawn dag therefore guarantees that $\mathrm{LCA_S}(u, v)$ is a P-node. □

Lemma 70 *Let u and v be strands in a performance dag, where v follows u in the serial execution order. If v is not a reduce strand and $\mathrm{LCA_S}(u, v)$ is a P-node, then $u \parallel v$.*

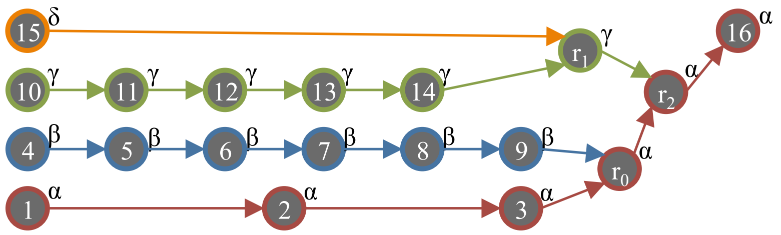


Figure 9-10: The view tree for the performance dag in Figure 9-5. Strands are labeled with either their label in Figure 9-2 or, for reduce strands, by their label r_0, r_1, or r_2 in Figure 9-5. Strands are additionally labeled with their view ID and colored to reflect their view ID.

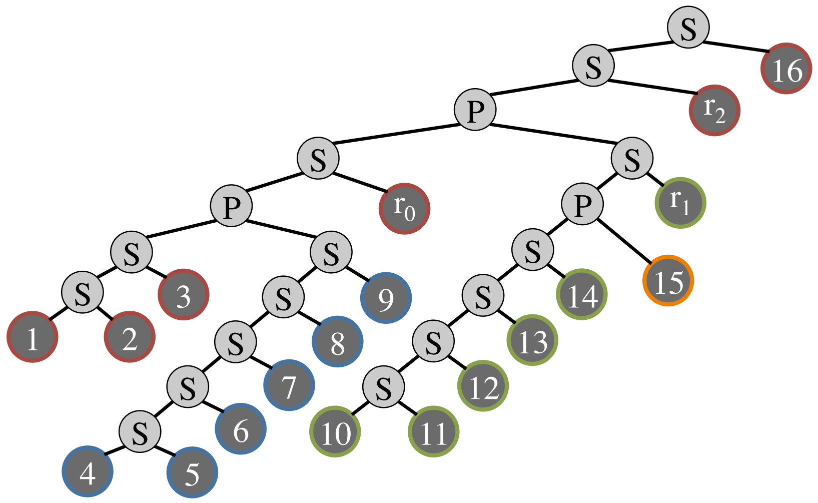


Figure 9-11: The view parse tree for the performance dag in Figure 9-5. Strands, which appear at the leaves of the tree, are labeled with either their label in Figure 9-2 or, for reduce strands, by their label r_0, r_1, or r_2 in Figure 9-5. Strands are colored to reflect their view ID.

Proof. Compared to the original performance dag, the spawn dag only removes edges that end at reduce strands, replacing an edge that enters reduce strand r with an edge into the sync strand responsible for r in the performance dag. Because v is not a reduce strand, if a path from u to v contained a reduce strand r in the performance dag, then it must also contain the sync strand responsible for r. Therefore, a path from u to v must therefore exist in the spawn dag as well. The lemma therefore follows from the analysis of SP parse trees by Feng and Leiserson, specifically, [134, Lemma 4]. □

For convenience, we show the following property of dags, which generalizes the property shown in [134, Lemma 6].

Lemma 71 *Suppose that three strands a, b, and c are encountered in order in a depth-first traversal of a dag G. If a path exists in G from a to b and no path exists from b to c, then no path exists from a to c.*

Proof. Assume for the purpose of contradiction that $v \prec w$. Because $u \prec v$, we have $u \prec w$ by transitivity, contradicting the assumption that $u \parallel w$. □

The view parse tree

The ***view parse tree*** captures which strands in a steal-specified Cilk computation can modify their views in parallel and which strands cannot. The view parse tree is derived from an alternative representation of the performance dag, called the ***view tree***, which is a directed tree that models the series-parallel relationships between the views. These series-parallel relationships are implied by the creation of views according to Invariant 2 and the destruction of views by REDUCE operations.

A view tree is constructed from the performance dag of a steal-specified Cilk computation by considering the strands of the dag in their serial execution order and applying the following rules in order:

1. A strand u for which $h(u)$ is a new, identity view is a leaf in the view tree. (Strand u is either the first strand in the computation or a continuation strand, by Invariant 2.)
2. If a strand u has the same view as the strand v immediately before it in the serial execution order — meaning that $h(u) = h(v)$ — then u is the sole successor of v in the view tree.
3. A reduce strand r that combines the views $h(u)$ and $h(v)$ has incoming edges from u and v in the view tree.

Figure 9-10 illustrates the view tree for the performance dag in Figure 9-5. Figure 9-10 illustrates two features of how view trees model reduce strands. First, in Rule 2, the strand u is never a reduce strand, because the view of the strand immediately before a reduce strand r in the serial execution order is always destroyed by r. Furthermore, a reduce strand r always has two predecessors u and v in the view tree, where u and v are the latest strands before r in the serial execution order with the views $h(u)$ and $h(v)$, respectively, that are reduced together by r.

The proof of correctness for SP+ relies on several properties, shown in the following lemma, on the structure of the view tree and its relationship to the performance dag.

Lemma 72 *Consider the view tree of a performance dag, and let G be a subtree of that view tree rooted at a strand v. Let u denote the first strand in G to execute in the serial order of execution. The following properties hold:*

(a) The views $h(u)$ and $h(v)$ are the same.
(b) There exists a path in the performance dag from any strand in G to v.
(c) There exists a path in the performance dag from u to any strand in G.
(d) Every path in the performance dag that begins at a strand outside of G and ends at a strand inside of G includes u.

PROOF. The lemma follows by induction on the construction of the view tree. The cases of a leaf strand, created by Rule 1, and a chain of strands created by repeated applications of Rule 2 are straightforward. We therefore focus on Rule 3.

In the case where a reduce strand r has incoming edges from two strands in the view tree, let G_1 and G_2 denote the two child subtrees of r, where G_1 executes before G_2 in the serial execution order. By construction of the view tree, the strands in G_2 execute immediately after the strands in G_1 in the serial execution order. Let h_1 and h_2 denote the two views reduced together at r, where h_1 and h_2 are adjacent and $h_1 > h_2$. Then the root of G_1 has view h_1 and the root of G_2 has view h_2. We justify each of the properties.

Property (a): By induction, the first strand in G_1 has the view h_1. Therefore, r shares the same view as the first strand in the subtree rooted at r.

Property (b): Because the roots of G_1 and G_2 both connect to r in the performance dag, by induction, there exists a path from any strand in either G_1 or G_2 to r.

Property (c): By induction, h_2 is the view of the first strand x in G_2. Because G_1 and G_2 are not connected in the view tree and all strands with the same view are connected in the view tree, no strand in G_1 shares the view h_2. By Invariant 2 (on reducer views), strand x must be the strand where h_2 is created, and therefore, x is a continuation strand of a spawn strand x' in G_1. Because, by induction, there exists a path from u to x', a path from x' to x, and a path from x to any strand in G_2, we know that there is a path in the performance dag from u to any strand in G.

Property (d): By the same argument above, any path in the performance dag into G_2 must therefore include strand x' in G_1. By induction, we thus have that any path from a strand outside of G_1 to a strand in either G_1 or G_2 must include the strand in G_1 that executes first in the serial execution order. □

The view parse tree represents the series-parallel composition of subtrees in the view tree, just as the SP parse tree represents a series-parallel dag. Figure 9-11 illustrates the view parse tree for the view tree in Figure 9-10. As Figure 9-11 illustrates, a strand u with a single predecessor v in the view tree is composed in series with the subtree rooted at v. If a reduce strand r has two predecessors u and v in the view tree, then the subtrees rooted at u and v are composed in parallel, and r is composed in series with the subtree modeling parallel composition.

The view parse tree thus captures the series-parallel relationships between views of a reducer. Consider two strands u and v in the performance dag, where u executes before v in the serial execution order. Strands u and v can modify their respective views in parallel, denoted $h(u) \parallel h(v)$, only if their least common ancestor in the view parse tree, denoted as $\mathrm{LCA_V}(u, v)$, is a P-node. Otherwise, the value of $h(v)$ reflects updates in the value of $h(u)$, denoted $h(u) \prec h(v)$, and u and v therefore do not operate on parallel views. The following lemma makes this intuition formal.

Lemma 73 *For two strands u and v in a performance dag, we have that $h(u) \parallel h(v)$ if and only if $\mathrm{LCA_V}(u, v)$ is a P-node.*

Proof. ($\Rightarrow$) Suppose for the purpose of contradiction that $h(u) \parallel h(v)$ and $\mathrm{LCA_V}(u, v)$ is an S-node a. Without loss of generality, suppose that u executes before v in the serial execution order. Let G_1 and G_2 denote the view subtrees corresponding to the two children of a, where G_1 contains u and G_2 contains v. By construction of the view parse tree, there exists a path from u to the root of G_1. Furthermore, because a is an S-node, G_2 contains only v and is composed in series with the root of G_1. Consequently, there exists a path from u to v in the view tree, contradicting the assumption that $h(u) \parallel h(v)$.

($\Leftarrow$) Assume for the purpose of contradiction that $h(u) \prec h(v)$ and $\mathrm{LCA_V}(u, v)$ is a P-node a. Let G_1 and G_2 denote the view subtrees corresponding to the two children of a, where G_1 contains u and G_2 contains v. Because $h(u) \prec h(v)$, there must exist a path from u to v in the view tree. By construction of the view parse tree, because a is a P-node, G_1 and G_2 must be two parallel subtrees in the view tree whose roots point to a common reduce strand r. There therefore exists a path in the view tree from one subtree of r to the other subtree of r, contradicting the fact that the view tree is a directed tree. □

We conclude this section with four lemmas on the structure of the view parse tree and spawn parse tree for a given performance dag, which the proofs in Section 9.7 use to show the correctness of SP+.

Lemma 74 *Let u, v, and w be three strands in a performance dag that execute in order in the serial execution order. Suppose that $h(u) \parallel h(v)$ and $h(v) \parallel h(w)$. Then we have $h(u) \parallel h(w)$.*

PROOF. In the view parse tree for the performance dag, let $a = \mathrm{LCA_V}(u,v)$ and $b = \mathrm{LCA_V}(v,w)$. Lemma 73 implies that both a and b are P-nodes. Because u, v, and w execute in order, one can observe that $\mathrm{LCA_V}(u,w)$ must be one of $\mathrm{LCA_V}(u,v)$ or $\mathrm{LCA_V}(v,w)$, and either way, $\mathrm{LCA_V}(u,w)$ is a P-node. Lemma 73 thus implies the lemma. □

Lemma 75 *Let u, v, and w be three strands in a performance dag that execute in order in the serial execution order. If $u \prec v$ and $u \parallel w$ and $h(u) \parallel h(w)$, then $h(v) \parallel h(w)$.*

PROOF. Assume for the purpose of contradiction that $h(v) \prec h(w)$. If $h(u) \prec h(v)$, then there is a path in the view tree from v to w, contradicting the fact that $h(u) \parallel h(w)$. If $h(u) \parallel h(v)$, then let w' be the first strand for which $h(w') \prec h(v)$. By construction of the view tree, strand w' is a continuation strand that executes between u and v in the serial execution order, and $h(u) \parallel h(w')$. Lemma 72 thus implies that $w' \prec w$ (Property (c)) and $u \prec w'$ (Property (d)). Hence, we have that $u \prec w$, contradicting the fact that $u \parallel w$. □

Lemma 76 *For two strands u and v in a performance dag, if $\mathrm{LCA_S}(u,v)$ is a P-node and $\mathrm{LCA_V}(u,v)$ is a P-node, then $u \parallel v$.*

PROOF. Without loss of generality, assume that v follows u in the serial execution order. Because Lemma 70 implies the lemma for non-reduce strands, suppose that v is a reduce strand. Assume for the purpose of contradiction that $u \prec v$ in the performance dag and $\mathrm{LCA_S}(u,v)$ and $\mathrm{LCA_V}(u,v)$ are P-nodes. Then Lemma 73 implies that $h(u) \parallel h(v)$. Let G denote the subtree of the view tree rooted at v. Because u executes before v in the serial execution order and $h(u) \parallel h(v)$, Lemma 72 implies that any path from u to v must contain the strand w, the first strand in G to execute in the serial execution order. Because w is not a reduce strand (by Invariant 2 on reducer views) and $u \prec w$, Lemma 70 implies that $\mathrm{LCA_S}(u,w)$ is an S-node. Furthermore, by construction of the spawn parse tree, reduce strand v occurs in the right subtree of $\mathrm{LCA_S}(u,w)$, meaning that $\mathrm{LCA_S}(u,w)$ must be an ancestor $\mathrm{LCA_S}(w,v)$. Consequently, $\mathrm{LCA_S}(u,v)$ is an S-node, contradicting our assumption that $\mathrm{LCA_S}(u,v)$ is a P-node. □

Lemma 77 *Suppose that three strands u, v, and w execute in order in the serial execution order of a performance dag. If $u \parallel v$ and $v \parallel w$ and $h(v) \parallel h(w)$, then $u \parallel w$.*

PROOF. If $u \parallel v$ and $v \parallel w$, then Lemma 69 implies that both $\mathrm{LCA_S}(u,v)$ and $\mathrm{LCA_S}(v,w)$ are P-nodes. One can show that either $\mathrm{LCA_S}(u,v)$ or $\mathrm{LCA_S}(v,w)$ is the least common ancestor of u and w in the spawn parse tree, meaning that $\mathrm{LCA_S}(u,w)$ is a P-node. If w is not a reduce strand, then Lemma 70 implies that $u \parallel w$.

Suppose instead that w is a reduce strand, and assume for the purpose of contradiction that $u \prec w$. Because $\mathrm{LCA_S}(u,w)$ is a P-node, by construction of the spawn dag, $u \prec w$ only when u is in series with an immediate predecessor of w other than the last immediate predecessor of w in the serial execution order. In the view tree, consider the child subtrees G_1 and G_2 of strand w, where G_1 executes before G_2 in the serial execution order. Because $\mathrm{LCA_S}(u,w)$ is a P-node and $u \prec w$, strand u must be in G_1. Meanwhile, the fact that $h(v) \parallel h(w)$ implies that v is in parallel with both G_1 and G_2 in the view tree. Consequently, v must come either before u or after w in the serial execution order, contradicting the fact that v executes between u and w in the serial execution order. □

9.7 Correctness of the SP+ algorithm

This section presents a proof that the SP+ algorithm correctly detects determinacy races in a steal-specified Cilk computation. For simplicity, this proof assumes that the Cilk computation operates on a single reducer. It is straightforward to extend the argument to handle more general cases. This section focuses on how SP+ detects races in a single steal-specified Cilk computation. Section 9.8 discusses how a polynomial number of such SP+ runs can provide the desired coverage for ostensibly deterministic Cilk programs, and Section 9.9 describes how steal specifications can be given inexpensively.

To formally argue that the SP+ algorithm correctly detects determinacy races, we relate the execution of the SP+ algorithm to the spawn and view parse trees for a steal-specified Cilk computation. From the construction of the spawn and view parse trees, each Cilk function invocation is represented by an assembly of strands and internal nodes in the spawn and view parse trees. Similarly to what's done in Section 9.4, we define the ***procedurification*** function $\mathcal{F}$ as the map from strands and nodes in these parse trees to Cilk function invocations.

We start with the following two lemmas that relate the execution of the SP+ algorithm to the structure of the spawn parse tree for a steal-specified Cilk computation. First, Lemma 78 shows that, when the SP+ algorithm executes a sync instruction in a function F, the P-stack of F consists of just one P-bag. Consequently, the action of the SP+ algorithm to move the contents of that P-bag into the S-bag of F is analogous to the action taken by the SP-bags algorithm at a sync strand. Using this result, Lemma 79 then relates the structure of the spawn parse tree to the contents of the S-bags and P-bags that SP+ maintains.

Lemma 78 *Consider the execution of* SP+ *on a steal-specified Cilk computation. When* SP+ *executes a sync strand in function F, the P-stack $F.P$ contains only a single P-bag.*

PROOF. The code for the SP+ algorithm in Figure 9-6 shows that two operations affect the view ID of the topmost P-bag in $F.P$ or the P-stack of any descendent function of F: executing a stolen continuation, which pushes a new P-bag with a new view ID on top of a P-stack, and calling REDUCE, which combines the top two P-bags in a P-stack. Because any view that is created at a continuation strand w in F are destroyed by a REDUCE operation r executed before the sync strand following w, any new P-bag pushed onto $F.P$ when SP+ executes w is popped off of $F.P$ when SP+ subsequently executes r. Consequently, when SP+ executes a sync strand in F, the P-stack $F.P$ contains only a single P-bag, namely, the P-bag it contained when F was invoked. □

Lemma 79 *Consider the execution of* SP+ *on a steal-specified Cilk computation. Suppose that strand u executes before strand v, and let $\mathcal{F}$ be the procedurification function mapping the spawn parse tree to Cilk function invocations. Let $a = \mathrm{LCA}_\mathrm{S}(u, v)$ be the least common ancestor of u and v in the spawn parse tree.*

- *If a is a P-node, then the procedure ID for $\mathcal{F}(u)$ belongs to a P-bag of $\mathcal{F}(a)$ when v is executed.*
- *If a is an S-node and v is not a reduce strand, then the procedure ID for $\mathcal{F}(u)$ belongs to the S-bag of $\mathcal{F}(a)$ when v is executed.*
- *If a is an S-node and v is a reduce strand, then the procedure ID for $\mathcal{F}(u)$ belongs to either the S-bag or the topmost P-bag in the P-stack of $\mathcal{F}(a)$ when v is executed.*

PROOF. The proof extends the argument in [134, Lemma 8], which relates the behavior of the SP-bags algorithm to the SP parse tree for a Cilk computation. We extend this argument to account for the reduce operations and reduce chains in spawn parse trees. We justify that, because the spawn parse tree is the canonical SP parse tree of the spawn dag and the SP+ and SP-bags algorithms are analogous with respect to non-reduce strands, then Lemma 8 in [134] completes the proof.

Suppose that a is in a reduce chain. By construction of the spawn parse tree, a is an S-node, v is a reduce strand in the right subtree of a, and $\mathcal{F}(u)$ is either $\mathcal{F}(a)$, a spawned subcomputation of $\mathcal{F}(a)$, or an invocation of REDUCE in the same reduce chain. If $\mathcal{F}(u) = \mathcal{F}(a)$, then the procedure ID for $\mathcal{F}(u)$ is stored in the S-bag $\mathcal{F}(a).S$. Otherwise the pseudocode in Figure 9-6 shows that, when $\mathcal{F}(u)$ returns, its procedure ID is placed in the topmost P-bag of the P-stack $\mathcal{F}(a).P$. Because reduce chains consist entirely of S-nodes, no stolen continuation executes in $\mathcal{F}(a)$ between the time that the procedure ID for $\mathcal{F}(u)$ is placed in a P-bag and the execution of v. Consequently, the procedure ID for $\mathcal{F}(u)$ is stored in the topmost P-bag of $\mathcal{F}(a).P$ when v is executed.

Suppose that a is in the spine or a sync block of $\mathcal{F}(a)$. By comparing the pseudocode of the SP+ algorithm in Figure 9-6 to that of the SP-bags algorithm [134], we observe that the SP+ algorithm moves procedure ID's among S-bags and P-stacks analogously to how the SP-bags algorithm moves procedure ID's among its S-bags and P-bags. In particular, Lemma 78 implies that, when a function F executes a sync strand, the P-stack $F.P$ contains just a single P-bag. The SP+ pseudocode executed at that sync strand therefore merges all procedure ID's in $F.P$ into $F.S$, equivalently to how the SP-bags algorithm moves the contents of its P-bag for F into its S-bag for F. Furthermore, the parent of any reduce chain in the spawn parse tree is a P-node. Therefore, for any strand v that executes after a reduce strand u, if $\mathrm{LCA}_{\mathrm{S}}(u, v)$ is this P-node, then the procedure ID for $\mathcal{F}(u)$ belongs to a P-bag in $\mathcal{F}(a)$ as expected.

Consider the execution of the SP-bags algorithm on the spawn dag of the steal-specified Cilk computation, whose SP parse tree is the spawn parse tree. By induction on the spawn dag, we conclude that, when a is in the spine or a sync block of $\mathcal{F}(a)$, if the SP-bags algorithm stores a procedure ID in an S-bag in $\mathcal{F}(a)$, then the SP+ algorithm stores the same procedure ID in an S-bag in $\mathcal{F}(a)$. Similarly, if the SP-bags algorithm stores a procedure ID in the P-bag in $\mathcal{F}(a)$, then the SP+ algorithm stores the same procedure ID in a P-bag in the P-stack of $\mathcal{F}(a)$. □

Next, the following lemma shows how the view ID's associated with the P-bags maintained by SP+ relate to the structure of the view parse tree.

Lemma 80 *Consider the execution of* SP+ *on a steal-specified Cilk computation. Suppose that strand u executes before strand v. Let $\mathcal{F}$ be the procedurification function mapping the view parse tree to Cilk function invocations, and let $a = \mathrm{LCA}_{\mathrm{V}}(u, v)$ be the least common ancestor of u and v in the view parse tree. Suppose that the procedure ID for $\mathcal{F}(u)$ belongs to a P-bag p in the P-stack $\mathcal{F}(a).P$ when v is executed. If a is an S-node, then $p.vid$, the view ID of p, matches the view ID of the topmost P-bag in $\mathcal{F}(v).P$. Similarly, if a is a P-node, then $p.vid$ does not match the view ID of the topmost P-bag in $\mathcal{F}(v).P$.*

PROOF. The code for the SP+ algorithm in Figure 9-6 shows that, when u is initially added to some P-bag in $\mathcal{F}(a).P$, it is added to the topmost P-bag in $\mathcal{F}(a).P$. Operations that affect the view ID of the topmost P-bag in $\mathcal{F}(a).P$ or the P-stack of any descendent function of $\mathcal{F}(a)$ include executing a stolen continuation, which pushes a new P-bag with a new view ID on top of a P-stack, and calling REDUCE, which combines the top two P-bags in a P-stack. Furthermore, calling or spawning a function G from a function F propagates the view ID of the topmost P-bag in G to that in F.

If $a = \mathrm{LCA}_{\mathrm{V}}(u, v)$ is an S-node, then $h(u) \prec h(v)$, meaning that between executing u and v, for every stolen continuation w that SP+ executes, SP+ executed a REDUCE operation that destroyed the view created at w. Consequently, between the time when u is added to the topmost P-bag in $\mathcal{F}(a).P$ and the execution of v, any P-bag pushed onto a function's P-stack (other than the first P-bag, which is always on the P-stack) is subsequently popped off. The topmost P-bag in $\mathcal{F}(v).P$ therefore has the same view ID as p.

If a is a P-node, then $h(u) \parallel h(v)$, meaning that between executing u and v, SP+ executed a stolen continuation to create a new view, but did not execute the REDUCE operation that destroys that view. Because all views created at continuation strands in a function are destroyed before a function returns, this stolen continuation must have been in a function G on the call stack between $\mathcal{F}(a)$ and $\mathcal{F}(v)$ inclusive. Consequently, the view ID of the topmost P-bag in G's P-stack does not match $p.\mathit{vid}$. Because $\mathcal{F}(v)$ is equal to or a descendant of G, the view ID of the topmost P-bag in $\mathcal{F}(v).P$ also does not match $p.\mathit{vid}$. □

Finally, we combine Lemmas 79 and 80 with the lemmas in Section 9.6 concerning the structure of the spawn and view parse trees to prove that the SP+ algorithm is correct.

Theorem 81 *The* SP+ *algorithm detects a determinacy race in a steal-specified Cilk computation A that uses a reducer if and only if a determinacy race exists.*

PROOF. Suppose that SP+ detects a determinacy race when executing a strand v. If v is a view-oblivious strand, then a determinacy race between v and a strand u that executes before v in the serial execution order requires only that $u \parallel v$. By Lemmas 69, 70 and 79, the SP+ algorithm correctly maintains this logical parallelism relation when v is a view-oblivious strand, and the theorem follows from the argument for the proof of correctness of the SP-bags algorithm [134, Theorem 10].

($\Rightarrow$) Suppose that v is a view-aware strand. Let $\mathcal{F}$ be the procedurification function mapping strands to Cilk function invocations. The pseudocode in Figure 9-7 shows that one of the following three cases occurs (where TOP gets the top-most element of the given P-stack without modifying that P-stack):

1. Strand v performs a `write` and $\mathit{reader}(\ell)$ belongs to a P-bag p where $p.\mathit{vid}$ does not equal $\mathrm{TOP}(\mathcal{F}(v).P).\mathit{vid}$.
2. Strand v performs a `write` and $\mathit{writer}(\ell)$ belongs to a P-bag p where $p.\mathit{vid}$ does not equal $\mathrm{TOP}(\mathcal{F}(v).P).\mathit{vid}$.
3. Strand v performs a `read` and $\mathit{writer}(\ell)$ belongs to a P-bag p where $p.\mathit{vid}$ does not equal $\mathrm{TOP}(\mathcal{F}(v).P).\mathit{vid}$.

In the first case, suppose that the ID in $\mathit{reader}(\ell)$ is set by a strand u, which executes before v. Because $p.\mathit{vid} \neq \mathrm{TOP}(\mathcal{F}(v).P).\mathit{vid}$, Lemma 80 shows that $\mathrm{LCA}_{\mathrm{V}}(u, v)$ must be a P-node, and therefore Lemma 73 implies that $h(u) \parallel h(v)$. If v is not a reduce strand, then because $\mathit{reader}(\ell)$ belongs to a P-bag p, Lemma 79 implies that $\mathrm{LCA}_{\mathrm{S}}(u, v)$ is a P-node. Otherwise, v is a reduce strand, and because $\mathit{reader}(\ell)$ belongs to a P-bag p such that $p.\mathit{vid} \neq \mathrm{TOP}(\mathcal{F}(v).P).\mathit{vid}$, Lemma 79 implies that $\mathrm{LCA}_{\mathrm{S}}(u, v)$ is a P-node. Either way, we have that both $\mathrm{LCA}_{\mathrm{S}}(u, v)$ and $\mathrm{LCA}_{\mathrm{V}}(u, v)$ are P-nodes, and therefore Lemma 76 implies that $u \parallel v$. Because both $u \parallel v$ and $h(u) \parallel h(v)$, a determinacy race exists between u and v. The other two cases are similar.

($\Leftarrow$) Now suppose that there exists a determinacy race in A on a location ℓ. Let u and v be two strands involved in such a race, where u executes before v and, if there are multiple such determinacy races, we choose the determinacy race for which the second strand executes earliest in the serial order. Suppose again that v is a view-aware strand. By definition of a determinacy race, we have that $u \parallel v$ and $h(u) \parallel h(v)$.

A determinacy race occurs in one of three ways:

1. Strand u writes ℓ and strand v reads ℓ.
2. Strand u writes ℓ and strand v writes ℓ.
3. Strand u reads ℓ and strand v writes ℓ.

In each case, let $\mathcal{F}$ be the procedurification function mapping the spawn and view parse trees to Cilk function invocations. We explicitly prove Case 3. The remaining cases are similar.

Suppose that u reads ℓ and v writes ℓ. When v is executed, let w be the strand such that *reader* (ℓ) stores the procedure ID of $\mathcal{F}(w)$. If $w = u$, then $u \parallel v$ implies that *reader* (ℓ) belongs to a P-bag p (by Lemmas 69 and 79), and $h(u) \parallel h(v)$ implies that $p.\mathit{vid} \neq \textsc{Top}(\mathcal{F}(v).P.\mathit{vid})$ (by Lemmas 73 and 80). Consequently, the pseudocode in Figure 9-7 shows that a determinacy race is reported. If $w \neq u$, then we consider the two cases of whether or not u updates *reader* (ℓ) when it executes.

If u updates *reader* (ℓ), then consider the sequence of updates to *reader* (ℓ) from the time u executes up to and including the time w executes. Let the strands performing the updates be $u_1, u_2, \ldots u_k$, where $u_1 = u$ and $u_k = w$. From the pseudocode in Figures 9-6 and 9-7, we have that for $i = 1, 2, \ldots, k-1$ that, when u_{i+1} executes, one of the following two cases applies.

- The procedure ID of $\mathcal{F}(u_i)$ is in an S-bag, in which case Lemmas 79 and 70 imply that $u_i \prec u_{i+1}$.
- Strand u_{i+1} is a reduce strand, the procedure ID of $\mathcal{F}(u_i)$ is in a P-bag p, and $p.\mathit{vid} == \textsc{Top}(\mathcal{F}(u_{i+1}).P).\mathit{vid}$. In this case, Lemma 80 implies that $\mathrm{LCA}_{\mathrm{V}}(u_i, u_{i+1})$ is an S-node, and therefore Lemma 76 implies that $u_i \prec u_{i+1}$.

In either case, we have $u_i \prec u_{i+1}$ for $i = 1, 2, \ldots, k-1$, which implies that $u \prec w$ by transitivity. Because $u \parallel v$, Lemma 71 implies that $w \parallel v$, and Lemmas 69 and 79 imply that w is in a P-bag p'. Furthermore, Lemma 75 implies that $h(w) \parallel h(v)$, and Lemmas 73 and 80 imply that $p'.\mathit{vid} \neq \textsc{Top}(\mathcal{F}(v).P).\mathit{vid}$. The pseudocode in Figure 9-7 therefore shows that a determinacy race is reported.

If u does not update *reader* (ℓ), then when u executes, we must have that the procedure ID of $\mathcal{F}(w)$ equals *reader* (ℓ) for some strand $w \parallel u$ that executes before u. Because $u \parallel v$ and $h(u) \parallel h(v)$, Lemma 77 implies that $w \parallel v$, and Lemmas 69 and 79 imply that w is in a P-bag p. Furthermore, Lemmas 71 and 74 imply that, regardless of whether $h(w) \prec h(u)$ or $h(w) \parallel h(u)$, we have that $h(w) \parallel h(v)$, and Lemmas 73 and 80 imply that $p.\mathit{vid} \neq \textsc{Top}(\mathcal{F}(v).P).\mathit{vid}$. The pseudocode in Figure 9-7 therefore shows that a determinacy race is reported. □

9.8 Analysis of the SP+ algorithm

This section discusses how the SP+ algorithm can be used to check if any execution on a given input of an ostensibly deterministic Cilk program that uses reducers contains a determinacy race involving a view-oblivious strand. If D is the maximum depth of nested spawns and K is the maximum number of continuations in any sync block, then we show that $\Omega(\max\{KD, K^3\})$ steal specifications are needed to elicit every possible view-aware strand, and $O(KD + K^3)$ steal specifications suffice. The proofs in this section can be adapted to construct these $O(KD + K^3)$ steal specifications.

The following theorem bounds the number of steal specifications needed to elicit all possible update strands. In a Cilk computation, if D is the Cilk depth and K is the maximum number of continuations in any sync block, then the following theorem implies that $O(KD)$ steal specifications are needed. The following theorem considers the Cilk computation's user dag, not its performance dag.

Theorem 82 *In a Cilk computation, all possible strands resulting from calls to* UPDATE *can be elicited in $\Theta(W)$ steal specifications, where W is the maximum number of continuations not followed by a sync strand in the same Cilk function along any path in the computation dag.*

PROOF. Consider the canonical SP parse tree for the user dag. Let a denote an internal node in this tree whose left child is l and whose right child is r. If a is an S-node, then the subcomputation under r inherits the value of the view $h(l)$. Because the reducer is a monoid, the value of $h(l)$ is the same, regardless of how the subcomputation under l was scheduled. The same situation holds if a is a P-node unless the subcomputation under r is stolen, in which case the subcomputation under r executes on a new, identity view. In this case, because the reducer is a monoid, the value of $h(r)$ does not depend on the computation executed before r.

For a strand u in the user dag, consider the root-to-u path p in the SP parse tree. From the argument above, the value of $h(u)$ depends only on the closest P-node $a \in p$ such that the right child of a inherits an identity view. The number of different values of $h(u)$ is therefore the number of P-nodes a in p for which u is in the right subtree of a.

A root-to-u path in the canonical SP parse tree passes through at most one sync block in each nested Cilk function F, and each P-node in F on that path corresponds to a continuation on the path to u in that sync block. Consequently, $\Omega(W)$ steal specifications are needed to elicit all possible update strands at the location of u. Because there exists a unique path in the SP parse tree from the root to each strand u, one can choose continuations to steal in a breadth-first manner, where two continuations w_1 and w_2 are stolen in the same specification if the same number of P nodes occur on the root-to-w_1 and root-to-w_2 paths in the tree. Consequently, $O(W)$ steal specifications suffice to elicit all possible strands resulting from calls to UPDATE. □

We now consider the number of steal specifications needed to elicit all possible reduce strands, assuming that the REDUCE operation is associative. Because a REDUCE operation always combines two adjacent views that have not yet been destroyed, given a sequence $\kappa = \langle k_1, k_2, \ldots, k_K \rangle$ of K views, every REDUCE operation on κ can be seen as combining two adjacent subsequences of κ. There are therefore $\binom{K}{3}$ distinct reduce strands that can be elicited on the views in κ, and therefore $O(K^3)$ specifications can elicit all possible reduce strands. The following theorem shows the lower bound that $\Omega(K^3)$ specifications are necessary to elicit every reduce strand.

Theorem 83 *Let $\kappa = \langle k_1, k_2, \ldots, k_K \rangle$ be an ordered set of K adjacent views. Any collection R of reduce trees on κ that contains each possible reduce strand at least once has size $|R| = \Omega(K^3)$.*

PROOF. To bound the number of reduce trees in R, let us characterize a REDUCE operation by the size of its larger input view. Each view h of a reducer that can be produced from combining views in κ corresponds to some subsequence of κ, and the ***size*** of h is the length of the subsequence corresponding to h. For example, a reduce strand that reduces the views represented by the subsequences $\langle k_a, k_{a+1}, \ldots, k_{b-1}\rangle$ and $\langle k_b, k_{b+1}, \ldots, k_{c-1}\rangle$ of κ reduces a view of size $b-a$ with one of size $c-b$. Let us consider reduce strands for which the size of its larger input is at least $n/2+1$.

To count the number of reduce trees containing such reduce strands, we imagine iteratively constructing the collection R of reduce trees by considering different view sizes in increasing order. For each size s, each view h of size s can be an input to multiple distinct possible reduce strands. Because $s \geq n/2+1$, each reduce tree in R can contain at most one view h of size s and at most one reduce strand r on such a view. A reduce tree in R that produces h might already contain r already; otherwise a new reduce tree must be added to R that contains r.

We can lower bound the number of reduce trees added to R for each size s using the following observations:

- There are $n-s+1$ distinct views of size s.
- For each view h of size s, there are $n-s$ distinct reduce strands that take h as an input.
- For each view h of size s, at most 2 reduce trees in R can produce h from a smaller view of a particular size s', where $n/2+1 \leq s' < s$. Consequently, there are at most $2(s-n/2-1)$ reduce trees already in R that contain distinct reduce strands on h.

These observations show that, for a particular size $s \geq n/2+1$, there are $(n-s+1)(n-s)$ different reduce strands on views of size s, and at most $(n-s+1)2(s-n/2-1)$ of these reduce strands can be exist in reduce trees already in R. For each size s, we must therefore add at least $(n-s+1)(2n-3s+2)$ new reduce trees to R. This bound holds as long as $2n-3s+2 > 0$, implying that $s < 2(n+1)/3$. Summing over the applicable sizes s, we have that

$$\begin{aligned} |R| &\geq \sum_{s=n/2+1}^{2(n+1)/3-1} (n-s+1)(2n-3s+2) \\ &= \Omega(n^3)\ . \end{aligned}$$

□

9.9 Rader

This section presents Rader, our prototype race detector that implements both the PEER-SET and SP+ algorithms. We evaluated Rader on six benchmarks. When running the PEER-SET algorithm, Rader incurs a geometric-mean multiplicative overhead of 2.56 (with a range of 1.01 to 6.65) over running the benchmarks without instrumentation. When running the SP+ algorithm, Rader incurs an overhead of 16.94 (with a range of 2.94 to 47.74). Both algorithms are implemented using compiler instrumentation, which accounts for some of this overhead. We measured the overhead of Rader over running the benchmarks with ***empty instrumentation***, that is, where each instrumented program point calls a function that simply returns. When running the PEER-SET algorithm, Rader incurs a geometric-mean multiplicative overhead of 2.31 (with a range of 1.00 to 4.64) over running the benchmarks with empty instrumentation. When running the SP+ algorithm, Rader incurs an overhead of 7.93 (with a range of 2.73 to 24.27). These averages are computed without including overhead for `ferret`, an outlier that has very little overhead, which we explain later in the section.

Implementation

The implementation of Rader consists of three parts: the library that implements the PEER-SET and SP+ algorithms, described in Sections 9.3 and 9.5; modification to the compiler to insert instrumentation that calls into the library; and modification to the Cilk runtime to execute a Cilk computation serially with steals and REDUCE operations dictated by a given steal specification.

We modified GCC 4.9 to insert instrumentation to identify parallel control constructs in the execution, akin to the Low Overhead Annotations [192] for Intel's Cilk Plus compiler. For instrumenting memory accesses, we piggyback on the ThreadSanitizer instrumentation [351], which has been supported in GCC since version 4.8 [385].

To implement the SP+ algorithm, Rader must execute a steal-specified Cilk computation, meaning that it must simulate steals according to the input steal specification. To accomplish this, Rader appropriately "promotes" various runtime data structures that would be modified if, after a worker executes the corresponding spawn, the continuation of the parent had been stolen [144]. When the worker resumes the parent later, it acts as if it has stolen the parent, and appropriately creates a new reducer view for the continuation. These promoted data structures also prompt the worker to check if it should execute any reduction.

Since Rader needs to check particular reductions according to the steal specification, the worker may need to hold off on a reduction instead of reducing eagerly, which is how Cilk runtime normally operates. We have modified the runtime so that the worker, when simulating steals, calls back to Rader to see if it should execute a reduction. Although the modified runtime no longer always performs reduction eagerly, we optimized the steal specifications that Rader uses as follows to use only constant space per steal.

Steal specifications

Although constructing the steal specification naively can cause the input to be as large as the computation dag, one can do better. Let D be the maximum depth of nested spawns, and let K be the maximum number of continuations in any sync block. Because Section 9.8 shows that $\Omega(\max\{KD, K^3\})$ executions are necessary to guarantee completeness and that $O(KD + K^3)$ suffice, no time is saved asymptotically if the system checks for more than one particular reduction or update per sync block. We therefore only need to make sure that Rader checks at least one reduction or update per sync block in a given execution. Consequently, the steal specification can be as simple as specifying which three continuations to steal in a sync block, to check REDUCE operations, or which continuations at a particular depth to steal, to check UPDATE operations. Each steal specification can steal the same continuations in every sync block, and the completeness guarantee still stands, as long as Rader is run with $O(K^3)$ different specifications. In practice, Rader takes as an input either three values specifying the continuations to be stolen, or a random seed and a value for K, in which case three different points are chosen randomly for each sync block. If a race is detected, Rader reports the labels corresponding to the stolen continuations that triggered the race, making it easy to repeat the run for regression tests.

Benchmark	*Description*	*Input size*
`collision`	$3D$ Collision detection	13 k pts, 264 k faces
`dedup`	File compression	*large*
`ferret`	Image similarity search	*large*
`fib`	Recursive Fibonacci	28
`knapsack`	Recursive knapsack	26
`pbfs`	Breadth-first search	$\|V\| = 2.5\,\mathrm{M}$, $\|E\| = 12.76\,\mathrm{M}$

Figure 9-12: Description of the benchmarks used, including input sizes.

					Running time with instr. (s)		
Benchmark	*Spawns*	*Syncs*	*Cilk frames*	*Mem. acc.*	*None*	*Cilk*	*Cilk + mem*
`collision`	1.67×10^4	7.94×10^4	5.22×10^4	1.97×10^8	0.603	0.607	1.437
`dedup`	9.40×10^4	3.00	9.40×10^4	8.39×10^8	11.596	11.577	12.467
`ferret`	2.58×10^2	3.00	2.59×10^2	3.82×10^4	8.058	8.108	8.142
`fib`	1.49×10^7	4.48×10^7	4.48×10^7	1.34×10^8	1.219	1.750	3.553
`knapsack`	3.53×10^6	3.96×10^6	7.06×10^6	2.37×10^8	0.459	0.527	1.387
`pbfs`	6.23×10^6	6.24×10^6	1.25×10^7	1.33×10^8	0.686	0.697	1.350

Figure 9-13: Basic measurements for benchmarks used, including execution characteristics, such as the number of `cilk_spawn` and `cilk_sync` statements, the number of Cilk frames created (which corresponds to the number of Cilk function instantiations), and the number of memory accesses. The *Running time with empty instr.* columns give the 1-processor running times in seconds of each benchmark with various amounts of empty instrumentation. The "*None*" column gives the running time of each benchmark when no instrumentation is inserted, which is the ordinary 1-processor running time of that benchmark. The "*Cilk*" column gives the running time of each benchmark with empty instrumentation only for Cilk's parallel constructs. The "*Cilk + mem*" column gives the running time of each benchmark with empty instrumentation for both Cilk's parallel constructs and memory accesses.

CPU	Intel Xeon E5-2665
Clock	2.4 GHz
Hyperthreading	Disabled
Turbo Boost	Enabled
Cores per processor chip	8
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	256 KiB
L3 cache/socket	20 MiB
DRAM	32 GiB
Operating system	Fedora 16, custom Linux kernel 3.6.11

Figure 9-14: Technical specifications of the machine used for benchmarking. The kernel was patched with support for thread-local memory mapping used in Cilk-M [238]. This patch was irrelevant to the experiment, and we do not believe it affects the numbers.

Experimental evaluation

We empirically evaluated Rader on the six benchmark applications described in Figure 9-12. Properties of these benchmark applications are described in Figure 9-13. We converted the pipeline programs `dedup` and `ferret` from the PARSEC benchmark suite [45] to use Cilk linguistics and a `reducer_ostream`, an output-stream reducer that is distributed with Intel Cilk Plus [196], to write its output. The synthetic `fib` benchmark uses a `reducer_opadd`,

Benchmark	PEER-SET	SP+ *(no steals)*	SP+ *(updates)*	SP+ *(reduce)*
`collision`	1.00	12.94	12.89	13.11
`dedup`	1.01	2.94	2.95	2.94
`ferret`	1.01	1.02	1.01	1.01
`fib`	6.65	16.99	16.99	44.33
`knapsack`	3.61	28.84	32.60	43.44
`pbfs`	4.52	44.22	44.47	47.74

Figure 9-15: Rader's overhead over running six benchmarks without instrumentation. The PEER-SET column shows the overhead over the one-processor running time without instrumentation when running the PEER-SET algorithm for checking view-read races only. The rest of the columns show the overhead when running the SP+ algorithm with different configurations. The "*no steals*" column corresponds to checking for determinacy races without eliciting any steals or REDUCE operations. The "*update*" column corresponds to checking for determinacy races with steals at middle continuation of every sync block. The "*reduce*" column corresponds to checking for determinacy races with randomly chosen steal points to elicit a subset of possible reductions.

Benchmark	PEER-SET	SP+ *(no steals)*	SP+ *(updates)*	SP+ *(reduce)*
`collision`	1.00	5.43	5.40	5.50
`dedup`	1.01	2.73	2.74	2.73
`ferret`	1.00	1.00	1.01	1.00
`fib`	4.64	5.83	5.80	15.21
`knapsack`	3.15	9.55	10.79	14.38
`pbfs`	4.45	22.48	22.60	24.27

Figure 9-16: Rader's overhead over running six benchmarks with empty instrumentation. The columns match those of Figure 9-15. Because PEER-SET does not use memory instrumentation, the overheads for PEER-SET are computed with respect to the "*Cilk*" running times in Figure 9-13. The SP+ overheads, meanwhile, are computed with respect to the "*Cilk + mem*" running times in Figure 9-13.

which is also part of Cilk Plus. All other benchmarks use user-defined reducers, including the bag data structure for `pbfs`, which is described in Chapter 3; a "hypervector" for `collision`; and a user-defined struct for `knapsack` [143]. Rader itself, including the modified runtime, and all of the benchmarks were compiled with `-O3` optimizations. Each running time is an average over 5 runs. Figure 9-14 summarizes the specifications of the benchmark machine used for all of the experiments.

Figure 9-15 presents the overhead of Rader over running each benchmark without instrumentation. As Figure 9-15 shows, the PEER-SET algorithm incurs little overhead. Because the overhead of PEER-SET mainly comes from creating and managing bags in the shadow frames upon spawns and syncs, benchmarks with high spawn, sync, and frame counts, such as `fib`, `knapsack`, and `pbfs`, exhibit slightly higher overhead.

For the SP+ algorithm, `fib`, `knapsack`, and `pbfs` also exhibit high overhead, because these benchmarks perform very little work per strand. Moreover, a large part of the work of these benchmarks involves accessing memory — stack memory, specifically, for `fib` and `knapsack` — which incurs overhead from instrumentation and accessing the shadow memory. Once we account for the instrumentation overhead (i.e., comparing overhead in Figure 9-15 and Figure 9-16), the overhead of SP+ for these benchmarks is reduced by a factor of 2 to 3. The `collision` benchmark falls somewhere in the middle, because it has many fewer spawns and syncs. Both `dedup` and `ferret`, on the other hand, incur very little overhead. Even though `dedup` performs many memory accesses, its running time is primarily dominated by file I/O. Finally, `ferret` is an outlier, both in terms of overhead and the number of instrumented events. It turns out that, among all of the library code that comes with PARSEC, `ferret` exhibits many determinacy races.[4] Because the reporting of races throws off timing due to printouts, we opted to instrument only the main `ferret` code without the rest of the library, meaning that only a small fraction of memory accesses within the computation are instrumented.

Let us look more closely at the overhead for SP+ for the three high-overhead benchmarks, `fib`, `knapsack`, and `pbfs`. SP+ incurs much higher overhead on `fib` and `knapsack` when checking races with randomly chosen steal points (which corresponds to the "SP+ *(reduce)*" column in Figures 9-15 and 9-16) compared with the other configurations, but running `pbfs` does not exhibit such a behavior. The reason for this behavior is that both `fib` and `knapsack` have very "small" sync blocks, specifically, each sync block contains essentially one spawn. Thus, randomly choosing the steal points in each sync block boils down to stealing almost every continuation, and thus the additional overhead incurred per sync block is significant compared to the work in the sync block. Running `pbfs` with randomly chosen steal points does not exhibit much more overhead compared to the other configurations because it has a sync blocks with as many as 21 continuations. Finally, SP+ with the other configurations does incur a much higher overhead on `pbfs` compared to `fib` and `knapsack`. This overhead comes from the relatively large memory footprint of `pbfs`. Even though all three benchmarks perform similar numbers of memory accesses, `fib` and `knapsack`, unlike `pbfs`, generally access stack space, which is frequently reused during a serial execution. The use of a shadow memory in SP+ exacerbates the large memory footprint in `pbfs`, and thus SP+ incurs many more cache misses when running `pbfs` compared to running `fib` and `knapsack`.

9.10 Related work

Race detection is a rich area actively being worked on. Roughly speaking, approaches to race detection either use static analysis [1, 7, 64, 131, 293, 319, 399] or dynamic analysis [89, 90, 115, 132, 137, 303, 318, 343, 398, 415]. We focus our discussion on the dynamic-analysis approach, the category our work falls under. In particular, we shall focus on related work that supports a similar language model, namely work on detecting determinacy races in programs with nested parallelism.

Nudler and Rudolph [302] proposed an ***English-Hebrew labeling*** scheme that labels "parallel tasks" in a computation based on two different traversal orders, such that comparing the labels suffice to tell whether the two tasks are logically in parallel. This scheme uses static labels, meaning that, once assigned, the labels do not change. The label size can grow proportionally to the maximum number of fork points in the program, that is, the number of execution points where parallel branches are spawned off.

Dinning and Schonberg [122] proposed a ***task-recycling scheme*** that improves upon the English-Hebrew labeling scheme by recycling labels for tasks, at the expense of failing to detect some races. They empirically demonstrated that the task-recycling scheme can be implemented efficiently.

Mellor-Crummey [280] proposed a different labeling scheme called ***offset-span labeling***, where the label sizes grow proportionally to the nesting depth, improving on the bound of the English-Hebrew labeling scheme. He also observed that, for parallel determinacy race

[4]We separately confirmed that these races exist using Intel's Cilkscreen race detector [197].

detection, it suffices to keep only two readers in shared memory, namely, the "left-most" and "right-most" parallel readers, which are the least and most recent reads in the serial execution order of the computation.

Feng and Leiserson proposed the SP-bags algorithm [134], which employs a disjoint-set data structure to maintain series-parallel relationships. SP-bags executes the computation serially and incurs near-constant overhead per check. They also observed that the logical parallel relationship is pseudotransitive, and thus it suffices to store only a single reader in the shadow memory.

Bender *et al.* proposed the SP-hybrid algorithm [37] that employs a scheme similar to English-Hebrew labeling, but manages the labels in a concurrent order-maintenance data structure, which allows for dynamic labeling and supports checks with constant overhead.

Raman *et al.* proposed ESP-bags [324] algorithm, which is similar to the SP-bags algorithm but extended to handle `async` and `finish` in Habanero-Java [82]. They subsequently proposed SPD3 detectors [325], also for Habanero-Java that maintains series-parallel relationships by keeping track of the entire computation tree, which has a simple implementation and executes in parallel.

Because our algorithms both extend the SP-bags algorithm and similarly use a disjoint-set data structure, they enjoy similar time and space bounds to SP-bags, with SP+ exhibiting additional overhead for simulating steals and reductions. Like SP-bags, however, they execute the computation serially. One distinct difference between our work and these algorithms is that SP+ handles race detection on computations with reducers, which correspond to non-series-parallel dags. To our knowledge, the SP+ algorithm is the first determinacy race detector that provides provable guarantees for computations that are not series-parallel. Nevertheless, the SP+ algorithm, albeit sound for a given execution, requires polynomial number of executions to guarantee complete coverage, due to the inherent nondeterminism in how the runtime manages reducers.

9.11 Conclusion

This chapter presented the PEER-SET and SP+ algorithms for detecting two unique types of races that arise from the incorrect use of reducer hyperobjects. Both algorithms are provably efficient and correct with respect to a given execution, and they incur modest overhead in practice. We have also shown that for an ostensibly deterministic Cilk program, polynomially many SP+ executions with different steal specifications suffice to elicit all possible view-aware strands, thereby providing the desired coverage. These algorithms, and the Rader tool that implements them, thus allows performance engineers to methodically detect programming errors that lead to nondeterministic behavior.

Both algorithms execute the computation serially, however, and a natural question is whether they can be parallelized to execute Cilk computations in parallel, so as to achieve better execution time for race detection. In particular, the PEER-SET algorithm has demonstrated negligible overhead when run serially, and an efficient parallel algorithm can lead to a light-weight always-on view-read race detection tool. Here, we lay out some of the challenges that we foresee in parallelizing these algorithms.

One challenge to parallelizing the PEER-SET algorithm is to figure out what minimal information needs to be stored in the shadow memory to correctly detect view-read races. The PEER-SET algorithm maintains the shadow memory to keep track of last readers in order to properly check whether two reads to a given reducer have the same peer set. If the algorithm executes the computation in parallel, there is no longer a clear notion of the last reader. For detecting determinacy races in parallel, Mellor-Crummey has demonstrated that it is sufficient to store only a "left-most" and a "right-most" reader [280]. Such a scheme works for detecting accesses that are logically in parallel, but it is unfortunately insufficient for checking for peer-set equivalence. Storing all parallel reads encountered, meanwhile, incurs non-constant space usage per reducer and time overhead per check.

The main challenge to an efficient parallel SP+ algorithm, on the other hand, is to achieve the desired time bound so that one can get speedup during parallel execution. Recall that the SP+ algorithm executes the computation according to a steal specification, which dictates what continuations to steal and what REDUCE operations to execute in what order. The constraints imposed by a steal specification can cause worker threads to be blocked at certain execution points, which can adversarially affect load-balancing. Conforming to the steal specification while maintaining good load balance seems to be an obstacle.

9.12 Recent developments

Since this work was originally published, some progress was made in the general domain of dynamic determinacy-race detection. Dimitrov *et al.* present a provably good algorithm for detecting determinacy races in programs whose computations can be modeled as 2D lattices [119]. Although a performance dag can be modeled as a 2D lattice, Cilk programs that use reducer hyperobjects exhibit additional complications to determinacy-race detection, including what qualifies as a determinacy race involving a view-aware strand and the nondeterminism of how the runtime system manages reducer views. Utterback *et al.* present a provably good and parallel determinacy-race detector [392] that can detect determinacy races in ordinary Cilk programs practically efficiently and in asymptotically optimal parallel running time.

Chapter 10

Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation

This chapter presents the Tapir compiler intermediate representation [347] for fork-join parallel programs supported by dynamic multithreading concurrency platforms. This work was conducted in collaboration with William Moses and Charles E. Leiserson.

10.1 Introduction

Mainstream compilers, such as GCC [369], ICC [198], and LLVM [232], now offer linguistic support for dynamic multithreading, such as is provided by the Cilk Plus [196] and OpenMP [26, 306] linguistic extensions for fork-join parallelism. But today's mainstream compilers that support fork-join parallelism fail to do a good job optimizing in the face of linguistic constructs for parallelism. As a result, parallelizing a program using dynamic multithreading incurs undue performance costs that undermine performance engineering efforts, causing performance engineers to struggle to realize the performance of theoretically efficient programs in practice.

Consider, for example, the parallel `cilk_for` loop on lines 4–5 in Figure 10-1, which indicates that iterations of the loop are free to execute in parallel. In the serialization of this loop, where the `cilk_for` keyword is replaced by an ordinary **for** keyword, each of the compilers GCC 5.3.0, ICC 16.0.3, and Cilk Plus/LLVM 3.9.0 observes that the call to `norm` on line 5 produces the same value in each iteration of the loop, and they optimize the loop by computing the value only once, before the loop executes. This optimization dramatically reduces the total time to execute `normalize`, from $\Theta(n^2)$ to $\Theta(n)$. Although this same optimization can, in principle, be performed on the actual parallel loop in the figure, none of these compilers performs this code-motion optimization. The same is true when the parallel loop is written using OpenMP.

This failure to optimize stems from how these compilers implement parallel linguistic constructs. To understand this problem, let us first review how compiler operate for serial languages. The compiler for a serial language, such as C [215] or C++ [376], can be viewed as consisting of three phases: a front end, a middle end, and a back end. The front end parses and type-checks the input program and translates it to an ***intermediate representation (IR)***, which typically represents the control flow of the program as a more-or-less language-

```
01 __attribute__((const)) double norm(const double *A, int n);
02
03 void normalize(double *restrict out, const double *restrict in, int n) {
04   cilk_for (int i = 0; i < n; ++i)
05     out[i] = in[i] / norm(in, n);
06 }
```

Figure 10-1: A function that GCC, ICC, and Cilk Plus/LLVM all fail to optimize effectively. The `cilk_for` loop on lines 4–5 allows each iteration of the loop to execute in parallel. The `norm` function computes the norm of a vector in $\Theta(n)$ time. The call to `norm` on line 5 can be safely moved outside of the loop.

independent ***control-flow graph (CFG)*** [8, Sec. 8.4.3]. In a CFG, a vertex denotes a ***basic block*** — a sequence of instructions with a single entry point for incoming branches and a single exit point for outgoing branches — and edges denote control flow between basic blocks. The middle end consists of many optimization passes that transform the IR into a more-efficient form. These optimizations tend to be independent of the instruction-set architecture of the target machine. The back end takes the optimized IR and translates it into machine code, performing low-level code-dependent optimizations.

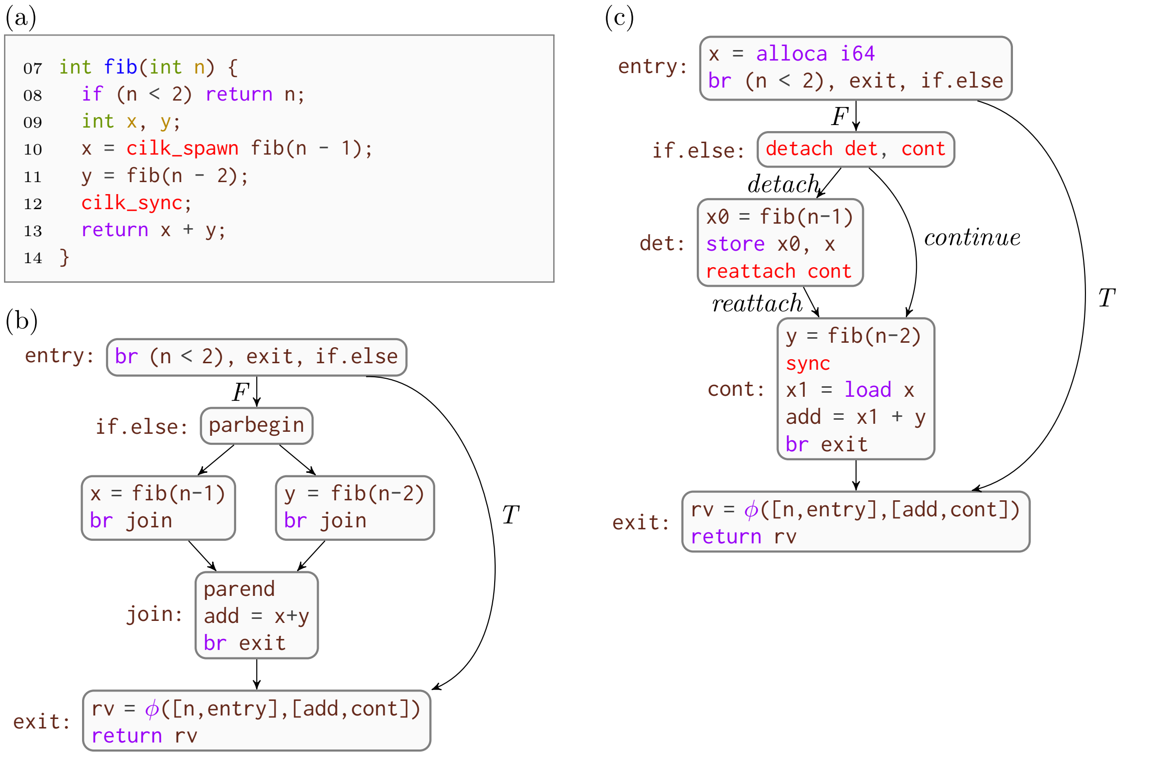
GCC, ICC, and Cilk Plus/LLVM all ***lower*** parallel constructs for dynamic multithreading in the front end — the front end reduces the parallel constructs to a more-primitive representation. To compile the loop in Figure 10-1, for example, the front-end translates the `cilk_for` loop into IR in two steps. First, the loop body (line 5) is lifted into a helper function. Next, the loop itself is replaced with a call to a library function implemented in the Cilk runtime system, which takes as arguments the loop bounds and the helper function, and handles the spawning of the loop iterations for parallel execution. Since this process occurs in the front end, it renders the parallel loop unrecognizable to middle-end loop-optimization passes, such as code motion. Thus, these compilers treat parallel constructs as syntactic sugar for opaque runtime calls, which confounds the many middle-end analyses and optimizations.

Previous research

This chapter aims to enable middle-end optimizations involving fork-join control flow by embedding parallelism directly into the compiler IR, an endeavor that has historically proven challenging [255, 256]. For example, it is well documented that traditional compiler transformations for serial programs can jeopardize the correctness of parallel programs [282]. In general, three types of approaches have been proposed to embed parallelism in a mainstream compiler IR.

First, the compiler can use annotations, called ***metadata***, to denote logical parallelism. The `parallel_loop_access` metadata in LLVM [259], for example, indicates that a memory access within a loop has no dependence on instructions in other iterations of the same loop. LLVM can only conclude that a loop is parallel if all its memory accesses are labeled with this metadata. Unfortunately, encoding parallel loops in this way is fragile, since a compiler transformation that moves code into a parallel loop risks serializing the loop from LLVM's perspective.

Second, the compiler can use a separate IR to encode logical parallelism in the program. Rather than embed parallelism into the IR, the HPIR [35, 420], SPIRE [216], and INSPIRE [211] representations, for example, model parallel constructs using an alternative IR, such as one based on the program's abstract syntax tree [8, Sec. 2.5.1]. Such an alternative IR



(a)

```
07 int fib(int n) {
08   if (n < 2) return n;
09   int x, y;
10   x = cilk_spawn fib(n - 1);
11   y = fib(n - 2);
12   cilk_sync;
13   return x + y;
14 }
```

Figure 10-2: Comparison of a traditional CFG with symmetric parallelism versus Tapir's CFG with asymmetric parallelism. **(a)** A Cilk function `fib`, which computes Fibonacci numbers. The `cilk_spawn` on line 10 allows the two recursive calls to `fib` to execute in parallel, and the `cilk_sync` on line 12 waits for the spawned call to return. A serial execution of `fib` executes `fib(n-1)` before `fib(n-2)`. **(b)** Illustration of the parallel flow graph [173, 364] of `fib`. Rectangles denote basic blocks, which contain C-like pseudocode for `fib`. Edges denote control flow between basic blocks. The `parbegin` and `parend` statements create and synchronize the parallel calls to `fib`. The `br` instruction either unconditionally branches to the named basic block or, based on the predicate, conditionally branches to either the first or second named basic block. The true and false edges from the conditional branch in `entry` are labeled T and F, respectively. The ϕ instruction, used to support a static single assignment (SSA) form [8, Sec. 6.2.4] of the program, takes as its arguments a pair associating a value with each predecessor of the current block. At runtime, the ϕ instruction returns the value associated with the predecessor basic block that executed immediately before the current basic block. **(c)** Illustration of the Tapir CFG for `fib` using the same format as (b). The `alloca` instruction allocates shared-memory storage for a local variable. Section 10.2 defines the `detach`, `reattach`, and `sync` instructions, as well as the *detach*, *reattach*, and *continue* edge types.

can support optimizations across parallel control flow without requiring changes to existing analyses and transformations for CFG's. But adopting a separate IR into a mainstream compiler has historically been criticized [257] as requiring considerable effort to engineer, develop, and maintain the additional IR for the same standards as the compiler's existing serial IR.

Third, the compiler can augment its existing IR to encode logical parallelism, which is the approach that Tapir follows. Unlike Tapir, all prior research on parallel precedence graphs [365, 366], parallel flow graphs [173, 364], concurrent control-flow graphs or "SSA" [242, 301], and parallel program graphs [341, 342] represent parallel tasks as symmetric entities in a CFG. For example, for the parallel `fib` function in Figure 10-2(a), the parallel flow graph in Figure 10-2(b) illustrates the symmetry of forked subcomputations.

Compiler component	*LLVM 3.8*	*Tapir/LLVM*	
Instructions	148,558	900	1,939
Memory behavior	10,549	588	
Optimizations	140,842	255	
Code generation	205,378	145	
Parallelism lowering	0	1,903	
New parallel optimizations	0	1,332	
Other	2,854,566	0	
Total	3,359,893	5,174	

Figure 10-3: Breakdown of the lines of code added, modified, or deleted in LLVM 3.8 to implement the Tapir/LLVM prototype.

Some of these representations struggle to handle common parallel constructs, such as parallel loops [216, 242], while others exhibit problems when subjected to standard compiler analyses and transformations for serial programs [173, 218, 242, 336, 341, 365, 366]. For example, the compiler analyses and optimizations in LLVM make an implicit assumption, which we call the ***lineage assumption***, for ordinary CFG's for serial programs that a basic block with multiple predecessors observes the variables of only one predecessor at runtime. Lee *et al.* observe, however, that parallel flow graphs break the lineage assumption [242]. For the parallel flow graph in Figure 10-2(b), for example, instructions in the `join` block must observe the values of `x` and `y` from both of its predecessors. Parallel loops exacerbate this problem by allowing a dynamic number of tasks to join at the same block. Previous research [4, 336] has proposed solutions to these problems, including additional representations of the program and augmented analyses that account for interleavings of parallel modifications to variables. Adopting these techniques into a mainstream compiler, however, seems to require extensive changes to the existing codebase.

The Tapir approach

This chapter introduces Tapir, a compiler IR that represents logical fork-join parallelism asymmetrically in the program's CFG. The asymmetry corresponds to the assumption of ***serial semantics*** [146], which means it is always semantically correct to execute parallel tasks in the same order as an ordinary serial execution.

Tapir adds three instructions — `detach`, `reattach`, and `sync`— to the IR of an ordinary serial compiler to express fork-join parallel programs with serial semantics. Figure 10-2(c) illustrates the Tapir CFG for the `fib` function. As with the symmetric parallel flow graph in Figure 10-2(b), Tapir places the logically parallel recursive calls to `fib` in separate blocks. But these blocks do not join at a synchronization point symmetrically. Instead, one block connects to the other, reflecting the serial execution order of the program.

Tapir's asymmetric representation of logically parallel tasks makes it relatively simple to integrate Tapir into an existing compiler's intermediate representation such as LLVM IR [259]. Figure 10-3 documents the lines of code added, modified, or deleted to implement Tapir/LLVM, a prototype of Tapir in LLVM. As Figure 10-3 shows, Tapir/LLVM was implemented with about 5000 lines, compared to LLVM's roughly 3-million-line codebase. Moreover, fewer than 2000 lines of code were needed to adapt LLVM's existing compiler analyses and transformations to Tapir.

The breakdown of lines is as follows. The lines for "Instructions" add Tapir's instructions to LLVM IR and adapt LLVM's routines for reading and writing LLVM IR and bitcode files. Conceptually, these changes allow LLVM to correctly compile a Tapir program to a serial executable with no optimizations. The lines for "Memory behavior" control how Tapir instructions interact with memory operations, preventing the compiler from creating any determinacy races in race-free codes. The lines for "Optimizations" perform any adjustments required for LLVM analyses and transformations to compile a Tapir program at optimization level `-O3`. Most of these modifications are not necessary for creating a correct executable but are added to allow the compiler to perform additional optimizations, such as parallel tail-recursion elimination as described in Section 10.5. The lines for "Code generation" fixed a bug in LLVM's implementation of `setjmp` which is independent of the implementation Tapir. The lines for "Parallelism lowering" translate Tapir instructions into Cilk Plus runtime calls and allow Tapir programs to be race-detected using a custom implementation of the SP-bags algorithm [134]. Finally, the lines for "New parallel optimizations" implement new optimization passes specifically for parallel code.

Tapir enables existing compiler optimizations to optimize across parallel control flow. The prototype implementation of Tapir/LLVM, for example, can move the call to `norm` in Figure 10-1 outside of the parallel loop, just as it can for the serialization of the loop. Tapir enables other optimizations, including common-subexpression elimination [290, Sec. 12.2], loop-invariant-code motion [290, Sec. 13.2], and tail-recursion elimination [290, Sec. 15.1], to optimize across parallel control flow and produce substantial performance improvements in practice. Tapir also enables new optimizations on parallel control flow. Tapir therefore allows the compiler to optimize a parallel program comparably to its serial counterpart, thereby helping performance engineers understand the performance of a parallel program in terms of the performance of its serial counterpart.

Tapir makes minimal assumptions about memory consistency [61, 321] for concurrent memory accesses. Tapir assumes only that memory is shared among all parallel tasks and that register state is local to each task. Although logically parallel tasks can access the same memory locations, Tapir does not specify semantics for concurrent accesses, and it does not restrict the compiler's optimizations based on any particular concurrency semantics. Because of its independence from any specific memory model, the Tapir approach is applicable to a variety of languages that support fork-join parallelism.

Tapir supports fork-join parallelism as expressed by parallel-language constructs such as those provided by Cilk and OpenMP. Although Tapir/LLVM happens to lower the parallel constructs to Intel's Cilk Plus runtime system, in principle, different lowering passes could be built to lower Tapir to other runtime systems, such as OpenMP's. By targeting the popular fork-join parallel models supported by various dynamic multithreading concurrency platforms, Tapir provides a language- and platform-independent way to embed parallelism into LLVM IR.

Cross-compiler comparison

Figure 10-4 compares the "work efficiency" of a suite of benchmark Cilk Plus programs when compiled with Tapir/LLVM versus the three mainstream Cilk compilers. For each of the four compilers, we evaluated each benchmark as follows. We compiled the benchmark with `-O3` optimizations and ran it on a single processor core, thereby measuring its work T_1. We also serialized each benchmark — replacing the Cilk linguistic constructs with their serial equivalents — to produce an equivalent serial C/C++ code, and then we compiled that code with each compiler using `-O3` optimizations and ran it on a single processor core to produce a ***serial running time*** T_S for the benchmark. We then computed the ***work efficiency***

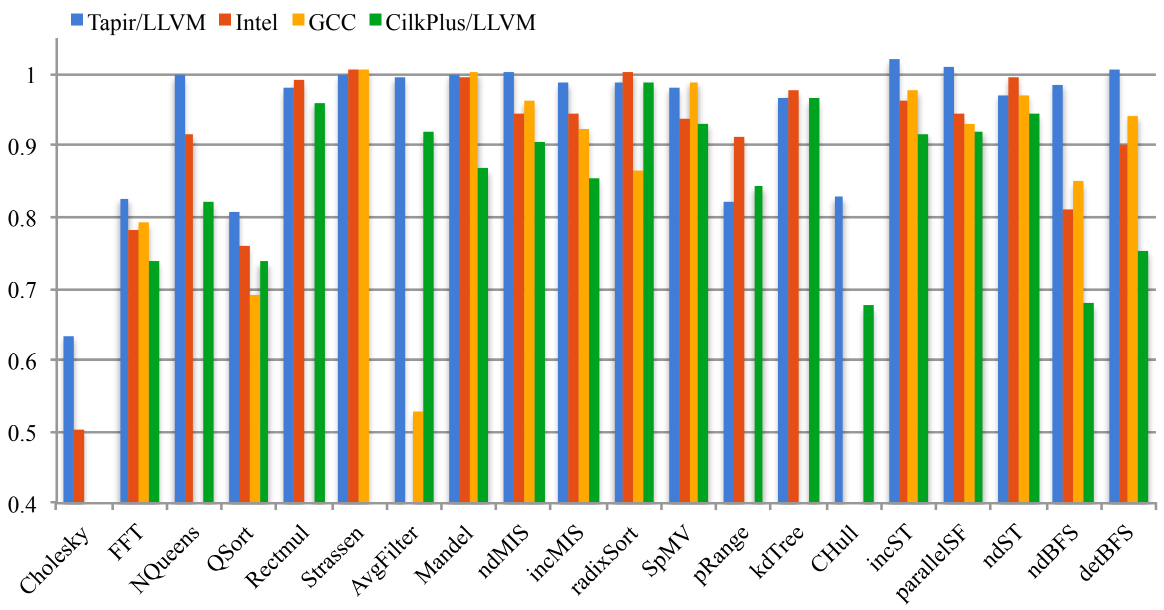


Figure 10-4: The work efficiency of mainstream compilers and Tapir/LLVM on the Cilk Plus benchmarks described in Section 10.7. For each compiler — Tapir/LLVM, ICC 16.0.3 (denoted as "Intel"), GCC 5.3.0 (denoted as "GCC"), and Cilk Plus/LLVM 3.9.0 (denoted as "Cilk Plus/LLVM") — the bar for each benchmark shows the ratio T_S/T_1, where the serial running time T_s is running time of the benchmark when the Cilk control constructs are replaced with their serial equivalents, and the 1-processor running time T_1 is the running time of the parallel Cilk code on 1 processor. A missing bar indicates that the compiler failed to correctly compile the code.

T_S/T_1 of the benchmark, which is plotted as a bar in Figure 10-4.

As the figure shows, Tapir/LLVM routinely exhibits higher work efficiency than the three mainstream compilers. A higher bar in the figure is better, because it means that the work of the parallel code exhibits less overhead compared to its serial equivalent. A value of 1.0 indicates perfect work efficiency — no overhead is incurred to enable parallel execution — while a value of 0.5 means that the work of the parallel code is twice that of its serial equivalent. Because the serial code avoids all the overheads introduced to support parallel execution, we normally expect the serial code to be faster than the parallel code on 1 processor, meaning that all bars should be at most 1.0. Due to the unpredictable nature of compilers, however, sometimes the 1-processor parallel execution outperforms the serial execution, as in the case of the incST benchmark, despite the overhead for parallelism.

This cross-compiler comparison of work efficiency can be misleading, however, as a compiler that produces more-optimized code may show a lower work efficiency than a compiler that produces less-optimized code, even if they have exactly the same overhead. For example, ICC tends to perform register allocation better than the other compilers, which translates to faster execution times for some benchmarks. Nevertheless, as Section 10.7 documents, the Tapir approach generally produces faster code than approaches that lower parallelism in the compiler front end.

Contributions

This chapter makes the following research contributions:

- The design of Tapir, a compiler IR for fork-join parallelism with asymmetric parallelism constructs that enables existing serial optimizations — such as common-subexpression

elimination, loop-invariant code motion, and tail-recursion elimination — to operate across parallel control flow, as well as allows for parallel optimizations.
- Denotational semantics for Tapir and a proof that, if an execution of a Tapir program exhibits no determinacy races, then it has serial semantics.
- The implementation of the Tapir/LLVM prototype compiler, which is implemented on top of the LLVM compiler by modifying only about 5000 source lines of code ($\sim$1.5 %).
- The implementation of parallel optimizations such as "unnecessary-synchronization elimination" and "parallel-loop spawning."
- Experiments which demonstrate that parallel programs exhibit lower overhead when compiled with Tapir/LLVM rather than other mainstream compilers.
- Experiments demonstrating the advantage of embedding fork-join parallelism into a compiler's IR, instead of handling parallel linguistics only in the compiler's front end.

Outline

The remainder of this chapter is organized as follows. Section 10.2 describes Tapir's representation and properties, as well as how Cilk and some OpenMP programs can be expressed using Tapir. Section 10.3 discusses how analysis passes can be adapted to operate on Tapir programs. Section 10.4 presents the denotational semantics of Tapir and argues for the serial semantics of determinacy-race-free Tapir programs. Section 10.5 describes various optimizations on parallel control flow that Tapir enables. Section 10.6 describes auxiliary software we developed to exercise and test Tapir/LLVM. Section 10.7 discusses our evaluation of the effectiveness of Tapir. Section 10.8 discusses related work. Section 10.9 concludes.

10.2 Tapir

This section describes Tapir and how it represents logically parallel tasks asymmetrically in the CFG of a program to support serial semantics. Tapir extends the compiler IR with three instructions — `detach`, `reattach`, and `sync`— to express fork-join parallel control flow. We describe these instructions and how they interact with static single-assignment (SSA) form [8, Sec. 6.2.4]. We describe how fork-join parallel language constructs from Cilk and OpenMP can be expressed using Tapir. Although we describe Tapir as an extension to LLVM IR [259], we see no fundamental reason why Tapir cannot be similarly implemented in other compilers.

Tapir instructions

Tapir extends LLVM IR with three instructions: `detach`, `reattach`, and `sync`. The `detach` and `reattach` instructions designate logically parallel tasks, and the `sync` instruction imposes synchronization on parallel tasks. To describe these instructions precisely, let $A = (V, E, v_0)$ be a CFG, where $v_0 \in V$ is the designated ***entry point*** of A where an execution of A must begin. Conceptually, the `detach` instruction is similar to a function call, and the `reattach` instruction is similar to a return. The three instructions have the following syntax:

`detach label` b`, label` c
`reattach label` c
`sync`

where b and c are (labels of) basic blocks in V. In LLVM IR terminology, both `detach` and `reattach` are ***terminator*** instructions that conclude the instructions in a basic block and implement outgoing control-flow edges.

A `detach` instruction takes a ***detached*** block b and a ***continuation*** block c as its arguments, and it allows b and c to operate in parallel. At runtime, a `detach` instruction in a block a terminates a and ***spawns*** a parallel task starting at b which can execute in parallel with the ***continuation*** block c. The CFG contains a ***detach*** edge $(a, b) \in E$ and a ***continue*** edge $(a, c) \in E$. The block b must be a single-entry block for which every exit from the sub-CFG reachable from b — the parallel task spawned by a — is terminated by a `reattach` whose continuation c matches the continuation in the corresponding `detach` instruction — the `detach` instruction ***reattached*** by the `reattach`. For each such `reattach` instruction, the CFG contains the ***reattach*** edge $(b', c) \in E$, where b' is the block terminated by the `reattach`.

In a sense, `detach` works like a function call to b that resumes at c after the subcomputation rooted at b returns, whereas `reattach` acts like a return. Unlike a function call and return, however, b is a block within the CFG of the function containing it, rather than a different function in the program, and b and c can operate in parallel. A `detach` does not require that b and c execute in parallel, but simply allows the runtime system to schedule them for parallel execution, if it so desires.

Tapir assumes that every CFG $A = (V, E, v_0)$ obeys the following invariants:

Invariant 84 *A* `reattach` *instruction reattaches one* `detach` *instruction.*

Invariant 85 *For each* `reattach` *instruction j that reattaches a* `detach` *instruction i, every path from v_0 to the block terminated by j passes through the detach edge of i.*

Invariant 86 *Every path starting from the detached block of a* `detach` *instruction i must reach a block containing a* `reattach` *instruction that reattaches i.*

Invariant 87 *If a path from a* `detach` *instruction i to a* `reattach` *instruction j that reattaches i passes through the detach edge of another* `detach` *instruction i', then it must also pass through a* `reattach` *instruction j' that reattaches i'.*

Invariant 88 *Every cycle containing a* `detach` *instruction i must pass through a* `reattach` *instruction that reattaches i.*

Invariant 89 *Any immediate successor of a* `reattach` *instruction cannot contain "ϕ instructions."*

These invariants imply that, at runtime, a `detach` instruction with detached block b and continuation c spawns the execution of the single-entry sub-CFG induced by the blocks on any path from b inclusive to c exclusive. We say that the `detach` instruction ***detaches*** this sub-CFG, and c is the ***continuation*** of the detached CFG.

Although memory state is shared among all parallel tasks in Tapir, it is organized as a tree of ***parallel contexts***. A new parallel context is created as a child of the current context when control enters a function or follows a detach edge. When control executes a `reattach` instruction or leaves a function the context is destroyed, and the parent's context becomes the current context. An `alloca` instruction allocates shared memory in the current context.

Tapir adopts LLVM's strategy for register availability. A register `x` defined in a basic block a is only available to subsequent instructions in a, but not to instructions before the definition. The register is only available within any basic block b if all paths from v_0 to b must pass through a, that is, a ***dominates*** b. As a result, a register defined in the CFG detached by a `detach` instruction is not available across reattach edges. That is, to communicate data out of a detached sub-CFG, the data must be transferred through shared memory and not through registers. In addition to preserving the fundamental invariants of LLVM, which simplifies the implementation of Tapir/LLVM, this behavior mimics what a runtime system must do more accurately than a design in which registers from multiple parallel tasks are all available after the tasks join.

At runtime, a `sync` instruction dynamically waits for the (dynamic) set of parallel tasks spawned within its parallel context and any of its descendant contexts to each execute their corresponding `reattach` instructions. We say that a `sync` instruction j ***syncs*** a `detach` instruction i if i and j belong to the same parallel context and the execution of the parallel task spawned by i might not have completed when j executes. The `detach` instructions that j might sync correspond to all `detach` instructions reachable in a reverse traversal of the CFG from j that does not pass through another `sync` instruction nor traverses a detach or a reattach edge.

Let us return to the Tapir CFG in Figure 10-2(c) and see how the three instructions are used to express the logical parallelism of the `fib` program in Figure 10-2(a). The `detach` instruction terminating the `if.else` block in Figure 10-2(c) allows blocks `det` and `cont` to execute in parallel. The `detach` instruction thus creates the detach edge $(\texttt{if.else}, \texttt{det}) \in E$ and the continue edge $(\texttt{if.else}, \texttt{cont}) \in E$. The `reattach` instruction in the `det` block reattaches the `detach` instruction in the `if.else` block, terminating the basic block `det` and creating the reattach edge $(\texttt{det}, \texttt{cont}) \in E$. The `sync` instruction in the `cont` block simply waits for the execution of the `det` block to complete. Unlike `reattach` instructions, `sync` instructions are not explicitly associated with `detach` instructions, and they, in fact, can be executed within conditionals.

Static single-assignment form

LLVM IR uses static single-assignment (SSA) form [8, Sec. 6.2.4], which must be adapted for Tapir programs. SSA form ensures that at most one instruction in a program function sets each register variable. LLVM IR employs the ***ϕ instruction*** [8, Sec 6.2.4] to combine definitions of a variable from different predecessors of a basic block. In adapting SSA to Tapir, a concern is that a ϕ instruction might allow registers defined in a detached sub-CFG to be used in its continuation. A basic block containing a ϕ instruction must avoid inheriting register definitions from predecessors that are connected by reattach edges. Otherwise, a register defined in a detached sub-CFG might not have been computed by the time the continuation executes.

We implement this constraint by simply forbidding reattach edges from going into basic blocks with ϕ instructions. But what if the continuation c of a `detach` instruction begins with a ϕ instruction? In this case, we create a new basic block c' containing only a branch instruction to c. We reroute the reattach and continuation edges originally going to c so that they go instead to c'. All other edges going to c are left in place.

The reason this solution works is as follows. No reattach edges in the resulting CFG go to blocks containing ϕ instructions. Because a detached sub-CFG does not dominate any outside block, registers in the detached sub-CFG can only be used in ϕ instructions of the immediate successors of the detached sub-CFG. Since the continuation is the only immediate successor of the detached sub-CFG and it contains no ϕ instructions, no registers

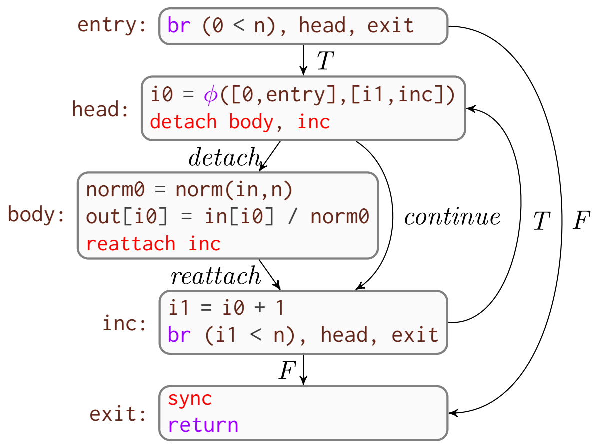


Figure 10-5: Tapir CFG for the parallel loop in Figure 10-1, using a similar format as the CFG's in Figure 10-2.

from the detached sub-CFG can be accessed in the continuation.

Asymmetry in Tapir

The `detach` and `reattach` instructions express parallel tasks asymmetrically both syntactically in the structure of the CFG and semantically in the way memory state is managed. Both asymmetries are illustrated by the example in Figure 10-2(c).

First, the CFG detached by a `detach` instruction is connected by a reattach edge to the continuation block of that instruction, even though they can execute in parallel. For example, the reattach edge between `det` and `cont` in Figure 10-2(c) breaks the symmetry between them. Reattach edges reflect the serial semantics of a Tapir program, which dictates that a serial execution of the program executes the detached CFG to completion before starting to execute its continuation. In contrast, parallel flow graphs and similar previously explored representations join logically parallel tasks in the CFG at a synchronization point. By supporting separate `reattach` and `sync` instructions, Tapir decouples the termination of a parallel task from its synchronization.

Second, although memory state is shared among all parallel tasks in Tapir, values in register variables are not guaranteed to be preserved across reattach edges. For example, the continuation block `cont` cannot assume that the register value `x0` returned by `fib(n-1)` in block `det` will be accessible, because the two blocks belong to different parallel contexts. Thus, `cont` must load it again after the `sync` instruction.

Section 10.3 shows how Tapir's asymmetric representation makes it easy for existing compiler analyses to work with Tapir CFG's.

Expressiveness of Tapir

Tapir can express logical fork-join parallelism in parallel programs with serial semantics. For example, it can express the parallelism encoded by the `cilk_spawn` and `cilk_sync` linguistics from Cilk++ [246] and Cilk Plus [196], as Figure 10-2 illustrates. Tapir can also concisely express the parallelism encoded by OpenMP `task` and `taskwait` clauses [26], as well OpenMP parallel sections [306]. Other parallel constructs can be represented as well, although operations that do not encode fork-join parallelism, such as OpenMP's `ordered` clause, cannot be represented directly using `detach`, `reattach`, and `sync` instructions.

Tapir can also express parallel loops, including `cilk_for` loops and OpenMP parallel loops that have serial semantics. Figure 10-5 illustrates Tapir's default representation of the parallel loop in Figure 10-1. As Figure 10-5 shows, Tapir can represent a parallel loop in the CFG as an ordinary loop, where the `head` block detaches the `body` block, and the `exit` block syncs all spawned executions of the `body` block. Section 10.5 describes how this representation of parallel loops allows existing compiler loop optimizations to operate on Tapir parallel loops with only minor modifications. Although this loop structure can exhibit poor parallel performance, when the loop body is small, separate optimization passes in Tapir/LLVM (see Section 10.5) transform this parallel-loop representation into alternative forms (see [277, Sec. 8.3], for example) that exhibit good performance.

10.3 Analysis passes

This section describes how LLVM's analysis passes can be adapted to operate on Tapir programs. We first discuss constraints on how Tapir programs can be safely transformed. We overview how LLVM's dominator analysis [8, Ch. 9], alias analysis [8, Ch. 12], and data-flow analysis [8, Ch. 9] were minimally modified to support Tapir.

Constraints on transformations

To be correct, a code transformation on a Tapir program must respect two properties: it must preserve the program's serial semantics and it must not introduce any new behaviors into the program's set of behaviors, although it may safely reduce the set of behaviors.

A program can exhibit more than one behavior if it contains a determinacy race [134]. A determinacy race can cause the program to execute nondeterministically depending on the order in which the two sections of code accesses the variable. Conceptually, to avoid introducing new behaviors, code transformations must not create determinacy races.

One way a code transformation can preserve the two correctness properties is by ***serializing*** a portion of the CFG: constraining parallel tasks to execute in their serial order, rather than allowing the runtime system to order things as it sees fit. An entire Tapir program can be serialized as follows:

- Replace every `detach` instruction by an unconditional branch to its detached block, which effectively removes the continue edge from the CFG.
- Replace every `reattach` instruction that reattach a `detach` instruction i with an unconditional branch to the continuation block of i.
- Remove all `sync` instructions.

The resulting program is called the ***serial elision*** [146] of the Tapir program, because all the parallelism has been elided. The serial elision contains no Tapir keywords and is a serial LLVM program that employs standard LLVM IR.

Alias analysis

Alias analysis [8, Ch. 12] determines whether different memory instructions might reference the same locations in memory. LLVM uses alias analysis to determine, for example, whether two memory instructions can be reordered. If they always reference different locations, it is generally safe to reorder them, but if they might refer to the same location, their order must be preserved. In addition to maintaining LLVM's invariants, code transformations in a Tapir program must avoid reordering instructions that introduce determinacy races.

To handle Tapir programs, alias analysis in LLVM treats `detach` and `sync` instructions as if they access memory. This adaptation involves examining the following four cases in which k is an arbitrary memory instruction, i is a `detach` instruction, and j is a `sync` instruction:

1. The instruction k moves from before i to after i.
2. The instruction k moves from after i to before i.
3. The instruction k moves from before j to after j.
4. The instruction k moves from after j to before j.

Neither Case 2 nor Case 3 can introduce a determinacy race, because both motions serialize the execution of k with respect to the sub-CFG spawned by i. Cases 1 and 4 might introduce a determinacy race, however, if k accesses a memory location that is also accessed by the CFG spawned by i. To avoid such problematic code motion, `detach` and `sync` are treated as instructions that access memory. Specifically, to handle Case 1, i is treated as if it were a function call that accesses all memory locations accessed in the CFG detached by i. Similarly, for Case 4, j is treated as if it were a function call that accesses all memory locations accessed by all instructions that j might sync. With these modifications, LLVM's alias analysis properly works on Tapir code.

Dominator analysis

Many optimization passes in the compiler middle end use ***dominator analysis*** [8, Ch. 9] to determine what basic blocks in the CFG must execute before or after what other basic blocks. Dominator analysis tells us when one instruction is guaranteed to execute before another instruction, and conversely, postdominator analysis tells us when one instruction is guaranteed to execute after another instruction.

Ideally, dominators and postdominators should work almost identically as in the serial elision of the Tapir program to ensure that serial code is optimized correctly. When given a parallel task, however, the analysis must understand that the spawned task is not guaranteed to execute before the continuation, and likewise that the continuation is not guaranteed to execute after the spawned task.

Ignoring the names of edges, the difference between the Tapir program's CFG $A = (V, E, v_0)$ and its serial elision's CFG $A' = (V, E', v_0)$ is the set $E - E'$ of continue edges, each of which connects a `detach` instruction to its continuation. A continue edge short-cuts the detached sub-CFG, changing the continuation's immediate dominator from the detached sub-CFG to the block containing the `detach` instruction itself. This configuration of detach, reattach, and continue edges looks much like an ordinary `if` construct in which the detached sub-CFG is conditionally executed. In a sense, for the sake of dominator analysis, the continuation can act as if the lineage assumption holds for its predecessors. As a result, LLVM's dominator analysis requires no changes to handle Tapir programs.

Data-flow analysis

Data-flow analysis [8, Ch. 9] examines the flow of data along different paths through a CFG $A = (V, E, v_0)$. A wide class of code transformations, including those that might move instructions across a reattach edge, rely on this analysis technique. Fundamental to data-flow analysis is an understanding of the set of possible program states at the beginning and end of each basic block $b \in V$, denoted $\text{IN}(b)$ and $\text{OUT}(b)$, respectively.

To illustrate how LLVM's data-flow analyses accomodate Tapir, let us examine the particular case of forward data-flow analysis; backward data-flow analysis is similar. In an ordinary serial CFG, forward data-flow analysis evaluates $\textsc{in}(b)$ as the union of $\textsc{out}(a)$ for each predecessor block a of b:

$$\textsc{in}(b) = \bigcup_{(a,b)\in E} \textsc{out}(a) \ .$$

To handle Tapir CFG's, data-flow analyses must be adapted specifically to handle continuation blocks. Because Tapir's asymmetric representation propagates register variables and memory state differently across a reattach edge, the modifications to LLVM data analyses consider registers and memory separately.

For variables stored in shared memory, the standard data-flow equations remain unchanged. Thus, LLVM need not be modified to handle them for Tapir.

For register variables, however, LLVM must be modified. In particular, LLVM's data-flow analyses must exclude the values in registers from an immediate predecessor a of a basic block b if the edge $(a, b) \in E$ is a reattach edge. Let E_R denote the set of reattach edges in E. For a Tapir CFG, forward data-flow analyses define $\textsc{in}(b)$ for register variables as

$$\textsc{in}(b) = \bigcup_{(a,b)\in E-E_R} \textsc{out}(a) \ ,$$

that is, they ignore predecessors across a reattach edge. With this change, Tapir/LLVM correctly propagates register variables through the CFG, never allowing register values in a basic block to use register values set in a logically parallel detached sub-CFG.

10.4 Denotational semantics for Tapir

This section presents formal, denotational semantics [348, 387] for Tapir. Although work has been done to develop formal semantics for LLVM [419], we examine the semantics of Tapir with respect to a simple serial programming language, in order to focus on the essential properties of these semantics. In particular, we show that, if an execution of a Tapir program contains no determinacy races, then it exhibits serial semantics, that is, the same semantics as its serial elision.

A simple integer language

We shall describe these semantics based on a simple serial programming language, Serial. Figure 10-6 presents the abstract syntax for Serial, which is defined on the following syntactic domains: $\mathcal{P}$, the domain of programs; Γ, the domain of statements; E, the domain of expressions; Ξ, the domain of variables; and $\mathcal{O}$, the domain of binary arithmetic operators. Serial is designed to illuminate the key semantic features of `detach`, `reattach`, and `sync`. In particular, in Serial, all accesses to memory are handled by `load` and `store` statements. Serial also supports the syntax $\{\Gamma\}$ for executing a statement within its own "scope."

The denotational semantics of Serial are straightforward. The state of a Serial program consists of a ***memory*** $M : \mathbb{Z} \to \mathbb{Z}$, which maps locations to values; and an ***environment*** $N : \Xi \to \mathbb{Z}$; which maps program variables to values. The semantics of a Serial program are defined by the ***expression evaluation*** function $\mathcal{E} : E \to N \to \mathbb{Z}$,

Syntactic domains	*Program production*	*Statement productions*
$\mathcal{P}$: Programs	$\mathcal{P} ::= \Gamma$	$\Gamma ::= \texttt{nop}$
Γ : Statements		$\mid \Xi = \texttt{load}\ E$
E : Expressions	*Expression productions*	$\mid \texttt{store}\ \Xi, E$
Ξ : Variables	$E ::= \mathbb{Z} \mid \Xi \mid \mathcal{O}\ E\ E$	$\mid \Xi = E$
$\mathcal{O}$: Binary arithmetic operators		$\mid \texttt{while}\ E\ \texttt{do}\ \Gamma$
		$\mid \Gamma_1; \Gamma_2$
		$\mid \{\Gamma\}$

Figure 10-6: Abstract syntax for the SERIAL language, which forms the basis in which we examine the semantics of Tapir.

```
$\mathcal{S}_S[\![x = \texttt{load}\ \epsilon]\!](M, N)$
1  return $(M, N[x / M(\mathcal{E}[\![\epsilon]\!](N))])$
```

```
$\mathcal{S}_S[\![\texttt{store}\ x, \epsilon]\!](M, N)$
2  return $(M[N(x) / \mathcal{E}[\![\epsilon]\!](N)], N)$
```

```
$\mathcal{S}_S[\![s_1; s_2]\!](M, N)$
3  return $\mathcal{S}_S[\![s_2]\!]\mathcal{S}_S[\![s_1]\!](M, N)$
```

```
$\mathcal{S}_S[\![\{s\}]\!](M, N)$
4  let $(M', N') = \mathcal{S}_S[\![s]\!](M, N)$
5  return $(M', N)$
```

Figure 10-7: Definition of the serial-statement-evaluation function $\mathcal{S}_S$ for `load` statements, `store` statements, statement sequencing ($s_1; s_2$), and statement scoping ($\{s\}$).

which maps the environment to an integer; and the ***serial-statement-evaluation*** function $\mathcal{S}_S : \Gamma \to M \times N \to M \times N$, which specifies how executing a statement transforms its state. An ***execution*** of a SERIAL program is the ordered set of program states it adopts when the statement evaluation function starts from a given input state and evaluates the statements of the program.

Figure 10-7 presents the definitions of $\mathcal{S}_S$ for `load` statements, `store` statements, statement sequencing, and statement scoping, which are pertinent to our description of Tapir's semantics. As Figure 10-7 shows, the behavior of $\mathcal{S}_S[\![\{s\}]\!]$ produces a new memory, but leaves the environment unchanged. This behavior resembles the behavior of a function call, which does not allow the local variables of the caller to escape the frame of the caller. We omit the definition of $\mathcal{S}_S$ for other statements, as well as the definition of $\mathcal{E}$, which are straightforward and familiar from other serial languages [387]. In particular, statements other than `load` and `store` do not read or write memory.

Semantics for Tapir

We describe the semantics for Tapir with respect to the semantics of SERIAL. We first define the syntax of a Tapir program as an extension of the syntax for SERIAL. In particular, a Tapir program supports the additional production rule for statements:

$$\Gamma ::= \texttt{detach}\ \Gamma_1\ \texttt{reattach}\ \Gamma_2 \mid \texttt{sync}$$

We also adjust the production rule for the Tapir program itself as follows:

$$\mathcal{P} ::= \Gamma; \texttt{sync}$$

To specify the semantics of a Tapir program, we augment the state of the program to

```
$\mathcal{S}_P[\![x = \texttt{load}\ \epsilon]\!](M, N, C, U)$
6  let $\ell = \mathcal{E}[\![\epsilon]\!](N)$
7  if $\ell \notin U(\mathrm{A}, \mathrm{P}).wr \cup U(\mathrm{D}, \mathrm{P}).wr$
8      return $(M, N[x/M(\ell)], C, U[(\mathrm{D}, \mathrm{S})/(U(\mathrm{D}, \mathrm{S}).rd \cup \{\ell\}, U(\mathrm{D}, \mathrm{S}).wr)])$
9  else
10     A determinacy race exists
```

```
$\mathcal{S}_P[\![\texttt{store}\ x, \epsilon]\!](M, N, C, U)$
11 let $\ell = \mathcal{E}[\![\epsilon]\!](N)$
12 if $\ell \notin U(\mathrm{A}, \mathrm{P}).wr \cup U(\mathrm{A}, \mathrm{P}).rd \cup U(\mathrm{D}, \mathrm{P}).wr \cup U(\mathrm{D}, \mathrm{P}).rd$
13     return $(M[\ell/N(x)], N, C, U[(\mathrm{D}, \mathrm{S})/(U(\mathrm{D}, \mathrm{S}).rd, U(\mathrm{D}, \mathrm{S}).wr \cup \{\ell\})])$
14 else
15     A determinacy race exists
```

Figure 10-8: Definition of the Tapir-statement-evaluation function $\mathcal{S}_P$ for `load` and `store` statements.

```
$\mathcal{S}_P[\![\texttt{detach}\ s_1\ \texttt{reattach}\ s_2]\!](M, N, C, U)$
16 let $(M', N', C', U') = \mathcal{S}_P[\![s_1]\!](M, N,$ new context,
       $U[(\mathrm{A}, \mathrm{S})/U(\mathrm{A}, \mathrm{S}) \cup U(\mathrm{D}, \mathrm{S})][(\mathrm{A}, \mathrm{P})/U(\mathrm{A}, \mathrm{P}) \cup U(\mathrm{D}, \mathrm{P})][(\mathrm{D}, \mathrm{S})/\emptyset][(\mathrm{D}, \mathrm{P})/\emptyset])$
17 return $\mathcal{S}_P[\![s_2]\!](M', N, C, U[(\mathrm{D}, \mathrm{P})/U(\mathrm{D}, \mathrm{P}) \cup U'(\mathrm{D}, \mathrm{S}) \cup U'(\mathrm{D}, \mathrm{P})])$
```

```
$\mathcal{S}_P[\![\texttt{sync}]\!](M, N, C, U)$
18 return $(M, N, C, U[(\mathrm{D}, \mathrm{S})/U(\mathrm{D}, \mathrm{S}) \cup U(\mathrm{D}, \mathrm{P})][(\mathrm{D}, \mathrm{P})/\emptyset])$
```

Figure 10-9: Definition of the Tapir-statement-evaluation function $\mathcal{S}_P$ for `detach` and `sync` statements.

keep track of parallel contexts and an "execution summary," which summarizes the locations in memory that are read and written in parallel.

To keep track of parallel contexts that arise in the evaluation of a Tapir program, each context is assigned a unique integer, $C : \mathbb{Z}$.

The execution summary maintains the logical series-parallel relationships among memory accesses in order to detect determinacy races. Formally, the execution summary is defined as follows. An ***access record*** $A \in 2^{\mathbb{Z}} \times 2^{\mathbb{Z}}$ is defined as a pair of sets of integers, where one set, denoted $A.rd$, is the ***read*** set, and the other, denoted $A.wr$, is the ***write*** set. We define an empty access record, denoted $\emptyset$, as the pair of empty sets. Access records are unioned element-wise, that is, for two access records A and A', the result of $A \cup A'$ is the pair $(A.rd \cup acc'.rd, A.wr \cup A'.wr)$. An ***execution summary*** $U : \{\mathrm{A}, \mathrm{D}\} \times \{\mathrm{S}, \mathrm{P}\} \to A$ maintains four access records. Conceptually, two access records $U(\mathrm{A}, \mathrm{S})$ and $U(\mathrm{A}, \mathrm{P})$, are maintained for "ancestor" parallel contexts, while two more, $U(\mathrm{D}, \mathrm{S})$ and $U(\mathrm{D}, \mathrm{P})$, are maintained for the current context and any of its descendants. The access records $U(\mathrm{A}, \mathrm{S})$ and $U(\mathrm{D}, \mathrm{S})$ record locations accessed logically in "series" with the current statement, while the access records $U(\mathrm{A}, \mathrm{P})$ and $U(\mathrm{D}, \mathrm{P})$ record locations accessed logically in "parallel."

To accomodate the augmented state of a Tapir program, we define the ***Tapir-statement-evaluation*** function $\mathcal{S}_P : \Gamma \to M \times N \times C \times U \to M \times N \times C \times U$. Figure 10-8 presents the definition of $\mathcal{S}_P$ for `load` and `store` statements, and Figure 10-9 present its definition for `detach` and `sync` statements. For all other statements, $\mathcal{S}_P[\![s]\!]$ produces the same memory and environment as $\mathcal{S}_S[\![s]\!]$ while leaving the context and execution summary unchanged.

In the definition of $\mathcal{S}_P[\![\texttt{detach}\ s_1\ \texttt{reattach}\ s_2]\!]$, we say that the new context C' created for $\mathcal{S}_P[\![s_1]\!]$ is a ***descendant*** of the input context C, and C is an ***ancestor*** of C'. Furthermore, the descendant/ancestor relationship between contexts is transitive.

An execution of a Tapir program is the partial order of program states it observes as $\mathcal{S}_P$ evaluates the program's statements starting from a given input state. We shall show how the execution of a Tapir program can be modeled as a series-parallel dag [134] A, where each edge denotes a program state, and the vertices denote evaluated statements. As Feng and Leiserson describe, a series-parallel dag has a designated source vertex and a designated sink vertex and can be constructed via recursive "series" and "parallel" compositions. The base case of the recursion is a single edge. A ***series*** composition between two disjoint dags A_1 and A_2 identifies the sink of A_1 with the source of A_2. A ***parallel*** composition identifies the source of A_1 with the source of A_2 and the sink of A_1 with the sink of A_2.

The following lemma shows that an execution of a Tapir program can be modeled as a series-parallel dag.

Lemma 90 *An execution of a Tapir program can be modeled as a series-parallel dag.*

PROOF. We first describe how an execution of a Tapir program can be modeled as a dag. Consider the evaluation $\mathcal{S}_P[\![s]\!]$ of each statement of the Tapir program, starting from the given input state. We separately consider `detach` statements, `sync` statements, and other statements in the program.

Consider a statement s other than a `detach` or a `sync` statement. If the evaluation $\mathcal{S}_P[\![s]\!]$ generates a new state from the given state, then it produces a new edge in the dag, which corresponds to the base case of a series-parallel dag construction. If the evaluation recursively evaluates a substatement, then the source of the dag modeling the recursive evaluation is unified with the sink of its input state, which corresponds to a series composition.

We describe the evaluation of `detach` and `sync` statements as producing ***pseudo series-parallel dags***, which are constructed recursively like series-parallel dags except that two such dags can be composed in parallel by either unifying their sources or their sinks, not necessarily both. We shall then argue that the dag produced by evaluating all statements in the Tapir program is in fact a series-parallel dag.

Consider the evaluation $\mathcal{S}_P[\![\texttt{detach}\ s_1\ \texttt{reattach}\ s_2]\!]$ on the state σ. Let A_1 denote the pseudo series-parallel dag modeling the recursive evaluation $\mathcal{S}_P[\![s_1]\!]$, and let A_2 denote the pseudo series-parallel dag modeling the recursive evaluation $\mathcal{S}_P[\![s_2]\!]$. The source of A_2 is unified with the source of A_1, producing a pseudo series-parallel dag.

Consider the evaluation of $\mathcal{S}_P[\![\texttt{sync}]\!]$ on state (M, N, C, U), which is modeled by the edge (u, v). Let $v_1, v_2, \ldots, v_n$ denote the sinks of previously generated states whose contexts match or are descendants of C. All sinks $v_1, v_2, \ldots, v_n$ are unified with the source u, producing a pseudo series-parallel dag.

We now argue, by induction over the dags generated in evaluating substatements of `detach` statements, that the dag model of the execution is in fact a series-parallel dag. Consider the evaluation of `detach` s_1 `reattach` s_2 on the state σ, and let A_1 denote the dag produced from evaluating $\mathcal{S}_P[\![s_1]\!]$. By induction, A_1 is a series-parallel dag. We consider two cases, depending on whether $\mathcal{S}_P[\![s_2]\!]$ evaluates a `sync` in the same context as the context of σ.

Suppose that $\mathcal{S}_P[\![s_2]\!]$ evaluates a `sync` statement in the same context as the context of σ, and let A_2 denote the dag produced by the evaluation of $\mathcal{S}_P[\![s_2]\!]$ up to the first such `sync`. By induction, A_2 is a series-parallel dag. The evaluation of the `detach` statement unifies the sources of A_1 and A_2 and the evaluation of the `sync` statement unifies their sinks. Hence A_1 and A_2 undergo a parallel composition of series-parallel dags, which produces a series-parallel dag.

Suppose that $\mathcal{S}_P[\![s_2]\!]$ never evaluates a sync statement in the same context as the context of σ. Let A_2 denote the dag produced by $\mathcal{S}_P[\![s_2]\!]$, which is a series-parallel dag by induction. The evaluation of the detach statement unifies the sources of A_1 and A_2. Because the final statement in a Tapir program is a sync statement in the same context as the input state, there must exist a sync statement in an ancestor context. The evaluation of such a sync statement unifies the sinks of A_1 and A_2. Hence A_1 and A_2 undergo a parallel composition of series-parallel dags, which produces a series-parallel dag. □

We define some terminology based on this dag model. Consider two distinct program states σ_1 and σ_2 in the execution of a Tapir program modeled by the dag A. We say that σ_1 precedes σ_2, denoted $\sigma_1 \prec \sigma_2$, if there is a path from σ_1 to σ_2 in A. If either $\sigma_1 \prec \sigma_2$ or $\sigma_2 \prec \sigma_1$, then σ_1 and σ_2 are in ***series***. Otherwise, σ_1 and σ_2 are in ***parallel***, denoted $\sigma_1 \parallel \sigma_2$.

Series-parallel dags admit the following properties [134, Lemma 2]:

Lemma 91 *Let G' be a series-parallel dag, let G be a series-parallel subdag of G', and let u and v be the source and sink of G, respectively. Then the following properties hold:*
- *There exists a path in G from u to any edge in G.*
- *There exists a path in G from any edge to v.*
- *Every path in G' that begins outside of G and enters G passes through u.*
- *Every path in G' that begins within G and leaves G passes through v.*

Proof of serial semantics

We now argue that a Tapir program that contains no determinacy races has the same semantics as its serialization. The function $\psi : \Gamma \to \Gamma$ maps statements to statements. The ψ function replaces serializes the given statement, replacing statements involving detach, reattach, and sync with their serial equivalents, that is,

$$\psi(\mathsf{detach}\ s_1\ \mathsf{reattach}\ s_2) = \psi(\{s_1\}; s_2)$$
$$\psi(\mathsf{sync}) = \mathsf{nop}$$

For all other statements, ψ trivially returns the statement produced by recursively running ψ on all substatements. The serialization of a Tapir program is the result of recursively applying ψ to every statement in the program.

Conceptually, determinacy-race-free Tapir programs exhibit serial semantics because, as long as $\mathcal{S}_P$ does not detect a determinacy race, then as $\mathcal{S}_P$ evaluates each statement of a Tapir program, it modifies the program state in a manner corresponding to how $\mathcal{S}_S$ modifies the program state as it evaluates the serialization of that statement. We must first argue that $\mathcal{S}_P$ maintains access records in a manner that detects a determinacy race if and only if one exists. Although $\mathcal{S}_P$ maintains access records in a manner that reflects the execution of the SP-bags algorithm [134] or the ESP-bags algorithm [324], Tapir programs are more general than the programs addressed by either of those algorithms, and the proofs of those algorithms do not directly apply. We therefore argue for the serial semantics of these programs by studying the behavior of $\mathcal{S}_P$ as it evaluates the program's statements.

The following lemma argues that the contents of the access records correctly identify locations that are read and written in parallel for detecting determinacy races.

Lemma 92 *Consider an execution A of a Tapir program. Let (M, N, C, U) denote a program state observed during the execution of A before a determinacy race is encountered. The following properties hold for each $\ell \in \mathbb{Z}$:*

(a) We have $\ell \in U(\mathrm{D}, \mathrm{P}).rd$ if and only if a `load` *statement in context C or a descendant thereof reads $M(\ell)$.*

(b) We have $\ell \in U(\mathrm{D}, \mathrm{P}).wr$ if and only if a `store` *statement in context C or a descendant thereof writes $M(\ell)$.*

(c) We have $\ell \in U(\mathrm{A}, \mathrm{P}).rd$ if and only if a `load` *statement in an ancestor context of C reads $M(\ell)$.*

(d) We have $\ell \in U(\mathrm{A}, \mathrm{P}).wr$ if and only if a `store` *statement in an ancestor context of C writes $M(\ell)$.*

PROOF. The proof follows by induction over the evaluation of statements in the program by $\mathcal{S}_P$. We consider `detach`, `sync`, `load`, and `store` statements in particular, because no other statements read or write memory or modify the execution summary.

Statement `detach` s_1 `reattach` s_2. Let $\sigma_0 = (M, N, C, U)$ be the state on which the `detach` statement is evaluated. We first consider the evaluation $\mathcal{S}_P[\![s_1]\!]$. Let σ_1 denote the state on which $\mathcal{S}_P[\![s_1]\!]$ is performed. As Figure 10-9 shows, σ_1 includes the execution summary U' in which

$$\begin{aligned} U'(\mathrm{A}, \mathrm{S}) &= U(\mathrm{D}, \mathrm{S}) \cup U(\mathrm{A}, \mathrm{S}) , \\ U'(\mathrm{A}, \mathrm{P}) &= U(\mathrm{D}, \mathrm{P}) \cup U(\mathrm{A}, \mathrm{P}) \text{ , and} \\ U'(\mathrm{D}, \mathrm{S}) &= \emptyset . \end{aligned}$$

Furthermore, σ_1 has a single immediate predecessor σ_0 in the dag A. Lemma 91 thus implies that, for any previously observed state σ, we have $\sigma \prec \sigma_1$ if and only if $\sigma \prec \sigma_0$, and otherwise $\sigma \parallel \sigma_1$. The context of σ_1 is also a new, descendant context of C. The properties therefore hold for σ_1.

Now consider the evaluation $\mathcal{S}_P[\![s_2]\!]$. Let $(M', N', C', U') = \mathcal{S}_P[\![s_1]\!]\sigma_1$. The properties hold for (M', N', C', U') by induction. As Figure 10-9 shows, $\mathcal{S}_P[\![s_2]\!]$ is performed on the state $\sigma_2 = (M', N, C, U'')$ in which

$$U''(\mathrm{D}, \mathrm{P}) = U(\mathrm{D}, \mathrm{P}) \cup U'(\mathrm{D}, \mathrm{S}) \cup U'(\mathrm{D}, \mathrm{P}) .$$

In the dag A, the state σ_2 has a single immediate predecessor σ_0. Lemma 91 thus implies that, for any previously observed state σ, we have $\sigma \prec \sigma_2$ if and only if $\sigma \prec \sigma_0$, and otherwise $\sigma \parallel \sigma_2$. In particular, all of the states observed in the recursive evaluation of $\mathcal{S}_P[\![s_1]\!]$ are in parallel with σ_2. Finally, states σ_0 and σ_2 share the same context. The properties hold thus for s_2.

Statement `sync`. Let $\sigma_0 = (M, N, C, U)$ be the state on which the `sync` statement is evaluated. Figure 10-9 shows that, if $\sigma_1 = \mathcal{S}_P[\![\mathsf{sync}]\!]\sigma_0$, then the execution summary U' of σ_1 satisfies

$$\begin{aligned} U'(\mathrm{D}, \mathrm{S}) &= U(\mathrm{D}, \mathrm{S}) \cup U(\mathrm{D}, \mathrm{P}) \text{ and} \\ U'(\mathrm{D}, \mathrm{P}) &= \emptyset . \end{aligned}$$

Let Σ denote the immediate predecessors of σ_1 in the dag A. Every state in Σ has a context equal to or descended from C. Furthermore, Lemma 91 implies that a previously observed state σ' precedes σ_1 if and only if σ' precedes some state in Σ, and otherwise $\sigma' \parallel \sigma_1$. The properties therefore hold for σ_1.

Load and store statements. Consider the evaluation of $\mathcal{S}_P[\![x = \texttt{load}\ \epsilon]\!]$ on state $\sigma_0 = (M, N, C, U)$. Figure 10-8 first shows that the evaluation first checks whether the location $\ell = \mathcal{E}[\![\epsilon]\!](N)$ is in $U(\mathrm{D}, \mathrm{P}).\mathit{wr}$ or $U(\mathrm{A}, \mathrm{P}).\mathit{wr}$. By induction, ℓ is in one of these sets only if a previously evaluated `store` statement produced a state that is logically in parallel with σ_0, which means that a determinacy race exists. If ℓ is in neither set, then Figure 10-8 shows that $\sigma_1 = \mathcal{S}_P[\![x = \texttt{load}\ \epsilon]\!]\sigma_0$ contains the execution summary U' where

$$U'(\mathrm{D}, \mathrm{S}).\mathit{rd} = U(\mathrm{D}, \mathrm{S}).\mathit{rd} \cup \{\ell\}\ .$$

Because σ_0 is the immediate predecessor of σ_1 in A, Lemma 91 implies that any other previously observed state σ precedes σ_1 if and only if it precedes σ_0, and otherwise $\sigma \parallel \sigma_1$. Hence the properties hold. The evaluation of a `store` statement is similar. $\square$

The next lemma justifies that, if an execution of a Tapir program exhibits a determinacy race, then $\mathcal{S}_P$ detects a determinacy race.

Lemma 93 *If an execution A of a Tapir program exhibits a determinacy race, then $\mathcal{S}_P$ detects a determinacy race.*

PROOF. Let s_1 and s_2 denote the two evaluated statements (vertices in A) involved in a determinacy race, where s_2 is the earliest statement involved in a determinacy race with a previously evaluated statement. Then s_1 and s_2 are logically in parallel. Let ℓ be the memory location accessed by both s_1 and s_2, and let (M, N, C, U) denote the state on which the evaluation $\mathcal{S}_P[\![s_2]\!]$ is performed. Suppose that s_2 is a `load` statement, which implies that s_1 is a `store` statement. Because s_1 is evaluated before s_2 and is logically in Lemma 92 implies that ℓ is in either $U(\mathrm{A}, \mathrm{P}).\mathit{wr}$ or $U(\mathrm{D}, \mathrm{P}).\mathit{wr}$. Consequently, as Figure 10-8 shows line 7 finds ℓ in one of these sets, and line 10 reports a determinacy race. The case where s_2 is a `store` is similar. $\square$

Finally, the following theorem shows that the execution of a Tapir program has serial semantics if it exhibits no determinacy races.

Theorem 94 *Suppose that the execution of a Tapir program exhibits no determinacy races. If s is the Tapir program and $s_0 = (M_0, N_0, C_0, U_0)$ is the initial program state, then $\mathcal{S}_P[\![s]\!]s_0 = \mathcal{S}_S[\![\psi(s)]\!](M_0, N_0)$.*

PROOF. The theorem follows by induction over the evaluation of the statements in the program. At each step, we consider the parallel evaluation $\mathcal{S}_P[\![s]\!](M, N, C, U)$ of a statement s evaluated on the program state (M, N, C, U) and compare it to the serial evaluation $\mathcal{S}_S[\![\psi(s)]\!](M_S, N_S)$ of $\psi(s)$ on the state (M_S, N_S). We justify that each step maintains two invariants:

Invariant 95 *The environments are equal: $N = N_S$.*

Invariant 96 *For every $\ell \in \mathbb{Z}$, if $\ell \notin U(\mathrm{A}, \mathrm{P}).\mathit{wr} \cup U(\mathrm{D}, \mathrm{P}).\mathit{wr}$, then $M(\ell) = M_S(\ell)$.*

The base case, the start of the program, holds trivially, because at the start of the program execution, the memories are the same, $M = M_S$; the environments are the same, $N = N_S$; and all access records in U are empty. We focus on the statements given in Figures 10-8 and 10-9, which modify memory or perform a `detach`, `reattach`, or `sync`, because $\mathcal{S}_P[\![s]\!]$ and $\mathcal{S}_S[\![\psi(s)]\!]$ perform identical operations on memory and the environment for all other statements, and $\mathcal{S}_P[\![s]\!]$ does not change U.

Statement x = `load` ϵ. The `load` statement is its own serialization. By the inductive hypothesis, we have $N = N_S$, and therefore $\mathcal{E}[\![\epsilon]\!](N) = \mathcal{E}[\![\epsilon]\!](N_S)$. Let $\ell = \mathcal{E}[\![\epsilon]\!](N)$. As the evaluation rules in Figures 10-7 and 10-8 show, when evaluating the `load` statement, neither $\mathcal{S}_S$ nor $\mathcal{S}_P$ modifies the memory, and $\mathcal{S}_P$ does not modify either $U(\mathrm{A},\mathrm{P}).\mathit{wr}$ or $U(\mathrm{D},\mathrm{P}).\mathit{wr}$. Therefore, Invariant 96 is maintained. Because the program contains no determinacy races, Lemma 93 implies that $\mathcal{S}_P$ does not reach line 10 in Figure 10-8. Consequently, line 7 implies that ℓ is not in $U(\mathrm{A},\mathrm{P}).\mathit{wr} \cup U(\mathrm{D},\mathrm{P}).\mathit{wr}$. Lemma 92 therefore implies all writes to $M(\ell)$ execute in series with the current statement. Hence, we have $M(\ell) = M_S(\ell)$. Both $\mathcal{S}_S$ and $\mathcal{S}_P$ therefore produce the same environment, $N[x/M(\ell)] = N_S[x/M_S(\ell)$, which maintains Invariant 95.

Statement `store` x, ϵ. Like the `load` statement, the `store` statement is its own serialization. The inductive hypothesis implies that $N = N_S$, and therefore $\mathcal{E}[\![\epsilon]\!](N) = \mathcal{E}[\![\epsilon]\!](N_S)$ and $N(x) = N_S(x)$. Let $\ell = \mathcal{E}[\![\epsilon]\!](N)$. The evaluation rules in Figures 10-7 and 10-8 show that evaluating s preserves the environment, and thus Invariant 95 is maintained. Because we assume that the program contains no determinacy races, Lemma 93 implies that $\mathcal{S}_P$ does not reach line 15 in Figure 10-8, and line 12 implies that ℓ is not in any of $U(\mathrm{A},\mathrm{P}).\mathit{wr}$, $U(\mathrm{A},\mathrm{P}).\mathit{rd}$, $U(\mathrm{D},\mathrm{P}).\mathit{wr}$, or $U(\mathrm{D},\mathrm{P}).\mathit{rd}$. By Lemma 92, all reads and writes to $M(\ell)$ execute in series before the current statement, which implies that $M(\ell) = M_S(\ell)$. Hence $\mathcal{S}_P$ and $\mathcal{S}_S$ produce $M[\ell/N(x)] = M_S[\ell/N_S(x)]$. The evaluation function $\mathcal{S}_P$, meanwhile, does not modify either $U(\mathrm{A},\mathrm{P}).\mathit{wr}$ or $U(\mathrm{D},\mathrm{P}).\mathit{wr}$. Hence Invariant 96 is maintained.

Statement `detach` s_1 `reattach` s_2. The serialization of `detach` s_1 `reattach` s_2 is $\{s_1\}; s_2$. Let (M, N, C, U) be the state on which the `detach` statement is evaluated. As Figure 10-9 shows, $\mathcal{S}_P$ first evaluates s_1. Let $(M', N', C', U') = \mathcal{S}_P[\![s_1]\!](M, N, \textbf{new}\text{ context}, U)$, and let $(M'_S, N'_S) = \mathcal{S}_S[\![s_1]\!](M_S, N_S)$. We first consider Invariant 96. The inductive hypothesis implies that Invariant 96 holds between M', U', and M'_S. Figure 10-9 shows that $\mathcal{S}_P$ ultimately produces $\mathcal{S}_P[\![s_2]\!](M', N, C, U'')$, where U'' only differs from U' in that it includes additional locations in $U'(\mathrm{D},\mathrm{P}).\mathit{wr}$. Therefore Invariant 96 is maintained among M', U'', and M'_S. We now consider Invariant 95. Figure 10-7 shows that $\mathcal{S}_S[\![\{s_1\}]\!](M_S, N_S)$ produces (M'_S, N_S), which implies that that

$$\begin{aligned}\mathcal{S}_S[\![\psi(\texttt{detach } s_1 \texttt{ reattach } s_2)]\!](M_S, N_S) &= \mathcal{S}_S[\![\{s_1\}; s_2]\!](M_S, N_S)\\ &= \mathcal{S}_S[\![s_2]\!] \circ \mathcal{S}_S[\![\{s_1\}]\!](M_S, N_S)\\ &= \mathcal{S}_S[\![s_2]\!](M'_S, N_S)\ .\end{aligned}$$

The inductive hypothesis implies that $N = N_S$, meaning that both $\mathcal{S}_P$ and $\mathcal{S}_S$ evaluate s_2 on the same environment. Invariant 95 therefore holds by induction on s_2.

Statement `sync`. The serialization of a `sync` is `nop`. By induction, we have that $N = N_S$,

(a)

```
15 void search(int low, int high) {
16   if (low == high) search_base(low);
17   else {
18     cilk_spawn search(low, (low+high)/2);
19     search((low+high)/2 + 1, high);
20     cilk_sync;
21   }
22 }
```

(b)

```
23 void search(int low, int high) {
24   if (low == high) search_base(low);
25   else {
26     int mid = (low+high)/2;
27     cilk_spawn search(low, mid);
28     search(mid + 1, high);
29     cilk_sync;
30   }
31 }
```

Figure 10-10: Illustration of common-subexpression elimination on a Cilk program. **(a)** The function `search` uses parallel divide-and-conquer to apply the function `search_base` to every integer in the closed interval [`low`, `high`]. **(b)** An optimized version of `search`, where the common subexpression `(low+high)/2` in lines 18 and 19 of the original version is computed only once and stored in the variable `mid` in line 26 of the optimized version.

and because neither $\mathcal{S}_S$ nor $\mathcal{S}_P$ modify the environment, Invariant 95 is maintained. Furthermore, the rule in Figure 10-9 shows that $\mathcal{S}_P[\![\texttt{sync}]\!]$, like $\mathcal{S}_S[\![\texttt{nop}]\!]$, preserves the memory. Figure 10-9 moves the locations in $U(\mathrm{D},\mathrm{P}).\,wr$ into $U(\mathrm{D},\mathrm{S}).\,wr$. By Lemma 92, all writes recorded in this set occur serially before the `sync` statement. Hence, for each $\ell \in U(\mathrm{D},\mathrm{P}).\,wr$, we have $M(\ell) = M_S(\ell)$, and therefore Invariant 96 is maintained. □

10.5 Optimization passes

Tapir enables LLVM's existing optimization passes [261] to optimize across parallel control flow. It also enables new optimization passes that specifically target Tapir's fork-join parallel constructs. This section discusses four representative optimizations. Common-subexpression elimination [290, Sec. 12.2] illustrates an optimization pass that "just works" with the additional Tapir instructions. Loop-invariant code motion [290, Sec. 13.2], and tail-recursion elimination [290, Sec. 15.1] were the only two out of LLVM's roughly 80 optimization passes that required any modification to work effectively on parallel code. Parallel-loop spawning serves as an example of a new optimization pass.

Common-subexpression elimination

The common-subexpression elimination (CSE) optimization identifies redundant calculations and transforms the code so that they are only computed once. For example, the expression `(low+high)/2` in Figure 10-10 is computed in both lines 18 and 19. Existing mainstream compilers that support fork-join parallelism fail to perform CSE on this code and compute `(low+high)/2` twice, because they cannot optimize across the `cilk_spawn` construct's associated runtime calls. Tapir/LLVM, however, performs this optimization. Like the vast majority of optimization passes in Tapir/LLVM, CSE "just works" on Tapir code without any modifications to LLVM's CSE pass.

Loop-invariant code motion

The loop-invariant code motion optimization (LICM) [290, Sec. 13.2] aims to move computations out of loop bodies if they compute the same value on every iteration of the loop. LICM is responsible, for example, for moving the call to `norm` in the parallel loop in Figure 10-1 outside of the loop, as described in Section 10.1. By adapting LICM to handle

(a)

```
32 void pqsort(int* start, int* end) {
33   if (begin == end) return;
34   int* mid = partition(start, end);
35   swap(end, mid);
36   cilk_spawn qsort(begin, mid);
37   qsort(mid+1, end);
38   cilk_sync;
39   return;
40 }
```

(b)

```
41 void pqsort(int* start, int* end) {
42   if (begin == end) return;
43
44   int* mid = partition(start, end);
45   swap(end, mid);
46   cilk_spawn qsort(begin, mid);
47
48   start = mid+1;
49   // Begin inlined code
50   if (begin == end) goto join;
51   mid = partition(start, end);
52   swap(end, mid);
53   cilk_spawn qsort(begin, mid);
54   qsort(mid+1, end);
55   cilk_sync;
56   goto join;
57   // End inlined code
58
59 join:
60   cilk_sync;
61   return;
62 }
```

(c)

```
63 void pqsort(int* start, int* end) {
64 pqsort_start:
65   if (begin == end) {
66     cilk_sync;
67     return;
68   }
69   int* mid = partition(start, end);
70   swap(end, mid);
71   cilk_spawn qsort(begin, mid);
72   start = mid+1;
73   goto pqsort_start;
74 }
```

Figure 10-11: Illustration of tail-recursion elimination on a parallel quicksort program. **(a)** A Cilk function `pqsort` that sorts an array of integers in the range specified by the `start` and `end` pointers. **(b)** A version of the Cilk function in (a), where the recursive tail call on line 37 has been replaced by one round of inlining. **(c)** A version of the Cilk function in (a), where tail-recursion elimination has removed the recursive tail call on line 37.

parallel loops in Tapir, Tapir/LLVM reduces the asymptotic work of this parallel loop from $\Theta(n^2)$ to $\Theta(n)$.

Tapir/LLVM requires a minor change to LLVM's LICM pass to handle parallel loops. Consider the CFG illustrated in Figure 10-5, which models the parallel loop in Figure 10-1. For the serial elision of the loop, which would have the same graph structure except with the continue edge missing, LLVM attempts to find candidate computations to move outside the loop by looking for instructions in the basic blocks of the loop body that dominate the exit block of the loop, in this case the block `inc`. (The block labeled `exit` is the exit of the function, not the loop exit.) For a parallel loop, however, this analysis fails to identify any code to move due to the existence of the continue edge. As Figure 10-5 shows, with the continue edge, blocks in the loop body can never dominate the exit block `inc` as they could for the serial elision.

Tapir/LLVM modifies LLVM's LICM pass to handle parallel loops by examining their serial elision, which essentially means ignoring continue edges. For simple parallel loop structures with a single continue edge, such as that shown in Figure 10-5, this modification was implemented by finding blocks in the loop body that dominate the predecessors of the loop exit. The modification required only 19 lines of LLVM's LICM pass to be changed.

Tail-recursion elimination

Tail-recursion elimination (TRE) [290, Sec. 15.1] aims to eliminate a recursive call at the end of a function, replacing it with a branch to the start of the function. By eliminating these recursive tail calls, TRE can avoid function-call overheads and reduce the stack space they consume. This optimization can especially benefit fork-join parallel programs, as many parallel runtime systems impose additional setup and cleanup overhead on a spawned function.

LLVM's existing TRE pass can perform the TRE optimization on Tapir programs with just a minor modification. Specifically the TRE pass ignores `sync` instructions after the tail-recursive call. Further, if TRE is applied and ignores a `sync` instruction, it must then insert a `sync` instruction before any remaining returns.[1]

To see why these `sync` instructions can be safely ignored, consider Figure 10-11, which illustrates how Tapir/LLVM's TRE pass operates on the `pqsort` function, a parallel version of Hoare's quicksort algorithm [186]. The original tail-recursive code is shown in Figure 10-11(a). Imagine inlining this tail recursive call as in Figure 10-11(b). For the inlined code, all `return` statements are replaced with branches to the `join` label. Because there is a `cilk_sync` at the start of `join`, the `cilk_sync` prior to the second return of the inlined code can be eliminated. Because this reasoning can be applied to inlining the tail-recursive call an arbitrary number of times, TRE can safely ignore a `sync` instruction after the final tail-recursive call, assuming that it inserts a `sync` instruction before all remaining returns. Modifying LLVM's TRE pass to support Tapir required only 42 lines to be changed in Tapir/LLVM.

Parallel-loop spawning

As discussed above and in Section 10.2, Tapir effectively represents parallel loops as a `for` loop whose body is spawned each iteration. Depending on the number of iterations of the loop and the amount of work inside each loop, however, spawning the iterations in this way can be inefficient. For a parallel loop with n iterations, spawning each iteration sequentially requires $\Omega(n)$ span, whereas spawning the iterations in a divide-and-conquer fashion [277, Sec. 8.3] requires $\Omega(\lg n)$ span. Although this distinction is significant when n is large, for small n, such as a small constant, the divide-and-conquer approach can perform poorly in practice due to function-call overheads.

Tapir/LLVM implements a brand new optimization pass that attempts to identify parallel loops and to transform them into divide-and-conquer loops when it seems beneficial. This pass was implemented in 1281 lines of code. The bulk of this code implements the work to identify parallel loops and to rewrite them to spawn iterations in a divide-and-conquer manner.

Other optimization passes

In addition, Tapir/LLVM implements two minor parallel optimization passes: unnecessary-synchronization elimination and puny task elimination. ***Unnecessary-synchronization elimination*** identifies and eliminates `sync` instructions that do not sync any `detach` instructions. ***Puny-task elimination*** serializes detached sub-CFG's that perform little to no work. In particular, if the running-time overhead of creating a parallel task outweighs the work in the task, the task might as well be run serially. Both of these optimization passes were implemented by augmenting LLVM's SimplifyCFG pass and required adding 51 lines of code.

[1] Unlike Cilk programs, a function in a Tapir program does not implicitly sync before it returns.

10.6 Auxiliary software

This section describes auxiliary software that we developed to exercise and test Tapir/LLVM. Although our research focuses on the middle end of the compiler and transformations on Tapir programs, we also implemented a front end for Cilk Plus and a pass to lower Tapir constructs to calls into the Intel Cilk Plus runtime. In addition, we developed compiler instrumentation that allows us to interface to a determinacy-race detector, to verify the correctness of the Tapir/LLVM implementation.

To create the front end, we created a modification of the Clang front end called PClang, which translates Cilk Plus codes to Tapir. PClang handles all the particular semantics of the Cilk Plus programming model, such as the implicit `cilk_sync` instructions that execute before returns and the implicit evaluation of the arguments of a spawned function before the spawn occurs. We also created a version of Clang that can handle some OpenMP codes.

Our lowering pass translates `detach`, `reattach`, and `sync` instructions into appropriate Cilk Plus runtime calls [191]. This pass lifts detached CFG's in the program into helper functions and inserts Cilk runtime calls to allow those helper functions to run in parallel. This pass, which amounts to approximately 1900 lines, allows us to run codes compiled with Tapir to verify that parallel executions produce correct results and achieve parallel speedup.

We implemented two tools to test the correctness of Tapir/LLVM. First, we modified LLVM's internal verification pass to ensure that Tapir invariants are maintained. Second, we augmented the lowering pass to optionally instrument `detach`, `reattach`, and `sync` instructions, as well as function entries and exits. This instrumentation allows us to use a custom implementation of the SP-bags algorithm [134] to check Tapir programs for determinacy races in a provably effective manner. This race detector allows us to quickly verify whether code transformations mistakenly generate determinacy races. We used these tools to test the correctness of Tapir/LLVM when run on the benchmarks from Section 10.7 and on many other codes. The tools helped us find bugs as we developed our Tapir/LLVM prototype, including bugs in LLVM itself, specifically within its code-generation passes. We also ran Tapir/LLVM against LLVM's internal regression suite and verified that Tapir did not produce any additional failures.

10.7 Evaluation

To evaluate the effectiveness of the Tapir approach, we evaluated our Tapir/LLVM prototype on 20 benchmarks. Although Tapir/LLVM performed well in the cross-compiler experiments shown in Section 10.1, those experiments merely gave us confidence in the viability of the Tapir approach. They do not show plainly from where Tapir's advantage accrues, because different compilers can produce dramatically different code, making it hard to ascertain whether the Tapir extensions to the IR are responsible for performance improvements or whether the improvements are due to something in the base compiler. For example, differences in the compiler's register-allocation algorithm alone can produce a 2× gap in performance, introducing large amounts of error into a comparative study. Moreover, seemingly small changes to the way a program is compiled can yield significant variations in performance.

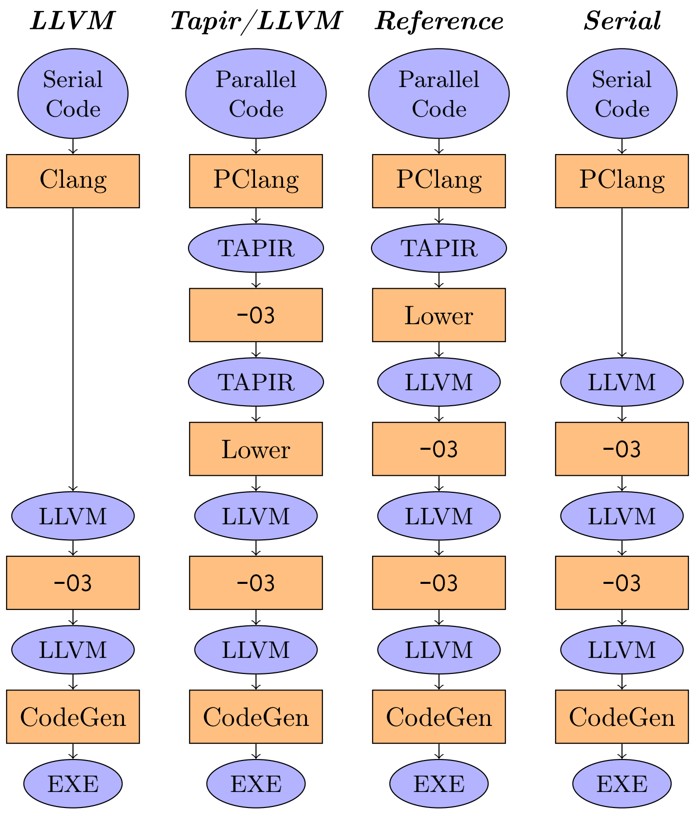


Figure 10-12: Comparing the compilation pipelines for LLVM, Tapir/LLVM, Reference, and Serial. Each block represents a compiler transformation, and each oval designates the format of the code at that point in a pipeline.

Compilation pipelines

Thus, instead of performing more cross-compiler comparisons in the style of Figure 10-4, this section seeks to conduct a more scientific evaluation. In particular, we focus on understanding how much of Tapir's performance improvement can be attributed to the approach of embedding parallelism in the IR rather than lowering parallelism in the compiler front end. Our study involves a single compiler pipeline which we modified to perform apples-to-apples comparisons. Figure 10-12 shows four pipelines based on the LLVM infrastructure that we used to conduct our experiments.

The first pipeline is for LLVM itself, which has the traditional three-phase structure. The Clang front-end takes serial C/C++ code and emits LLVM IR. The `-O3` middle-end optimizes the IR, and the CodeGen back-end lowers LLVM IR to the machine code for a particular hardware platform. The `-O3` pass is actually the Tapir/LLVM version, but it doesn't really matter, since for LLVM IR as input, the Tapir/LLVM version operates identically to the unmodified LLVM version.

The second pipeline shows how Tapir/LLVM is organized. The pipeline takes parallel Cilk Plus code as input, and the PClang front end emits Tapir. The middle-end now consists of three steps: `-O3` optimization, a Lower pass to lower Tapir to LLVM IR, and another pass at `-O3` optimization. The first `-O3` pass performs optimizations on the Tapir representation. The lowering pass translates all the Tapir-specific constructs to LLVM IR, including calls to the Intel Cilk Plus runtime. The second `-O3` pass performs optimizations on the LLVM IR, including optimizations on the function calls inserted by the lowering pass. Finally, the

Suite	*Benchmark*	*Description*
Cilk	Cholesky	Cholesky decomposition, 4000×4000 matrix, 8000 nonzeros
	FFT	Fast Fourier transform, 20 M composites
	NQueens	n-Queens problem, $n = 13$
	QSort	Parallel quicksort, 50 M elements
	RectMul	Cubic-time multiplication of 4096×4096 and 4096×2048 matrices
	Strassen	Strassen matrix multiplication, 4096×4096 matrices
Intel	AvgFilter	Averaging filter over an image
	Mandel	Mandelbrot set computation
PBBS	ndMIS	Nondeterministic maximal independent set algorithm
	incMIS	Incremental maximal independent set algorithm
	radixSort	Radix sort
	SpMV	Matrix-vector multiply using compressed sparse rows
	pRange	Range operations on a suffix array
	kdTree	Parallel k-d tree
	CHull	Convex hull
	incST	Incremental spanning tree algorithm
	parallelSF	Spanning forest
	ndST	Nondeterministic spanning tree algorithm
	ndBFS	Nondeterministic BFS algorithm
	detBFS	Deterministic BFS algorithm

Figure 10-13: Descriptions of the 20 application benchmarks used to evaluate Tapir. The "*Cilk*" benchmarks were taken from the MIT Cilk-5 benchmark suite [146]. The "*Intel*" benchmarks were taken from the collection of Intel Cilk Plus example programs [189]. The "*PBBS*" benchmarks were taken from the Problem-Based Benchmark Suite [360].

CodeGen back end lowers LLVM IR to machine code.

To study the effectiveness of the Tapir approach, we could compare the work a parallel benchmark compiled with Tapir/LLVM to the running time of its serial elision compiled with LLVM. This comparison would not be fair to LLVM, however, because it only executes one `-O3` pass and Tapir/LLVM executes two. Although one might think that a second pass of `-O3` would be redundant, it is not. For example, a simple matrix-multiplication code runs 13% faster after two rounds of optimization compared to just one. Moreover, although many benchmarks run faster after two `-O3` passes, some actually do run slower.

A fairer comparison is to execute the serial elision on the fourth pipeline Serial, which executes two `-O3` passes, just like Tapir/LLVM. In fact, this comparison is essentially the same as running the serial elision of a benchmark through Tapir/LLVM, because the PClang front end operates identically to Clang on serial code, and the Lower pass is a no-op for serial code.

The third pipeline, called Reference, models how mainstream compilers work today, where parallel language constructs are transformed into runtime calls before any optimization can take place. The only difference between Reference and Tapir/LLVM is that the order of the first `-O3` pass and the Lower pass are switched. Although Reference lowers the parallel constructs early, to ensure that the comparison is fair, two iterations of `-O3` are executed. Additionally, Reference is implemented on the same codebase as Tapir/LLVM. Because it runs the identical compiler with identical optimization passes, Reference is directly comparable to the Tapir/LLVM implementation. Thus, the Tapir/LLVM and Reference pipelines allow us to compare the impact of the Tapir strategy with the mainstream compiler strategy keeping all — well perhaps, many — other things equal.

CPU	Intel Xeon E5-2666 v3
Clock	2.9 GHz
Hyperthreading	Disabled
Turbo Boost	Disabled
Cores per processor chip	9
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	256 KiB
L3 cache/socket	25 MiB
DRAM	60 GiB

Figure 10-14: Technical specifications of the Amazon `c4.8xlarge` spot instance used for benchmarking. The system was quiesced to permit careful measurements by turning off Turbo Boost, dvfs, hyperthreading, extraneous interrupts, etc.

Benchmarking

To evaluate the compiler pipelines, we assembled a collection of benchmark programs taken from the MIT Cilk benchmark suite [146], Intel Cilk code samples [189], and the Problem-Based Benchmark Suite [360]. From these benchmark suites, we selected stable benchmark programs that tend to exhibit little performance difference when the number or order of optimization passes is changed. Figure 10-13 describes the benchmarks tested in detail.

We compiled each program in our benchmark suite with both Tapir/LLVM and Reference, and we ran them on both 1 and 18 cores of our test machine. Additionally, we ran the compilers on the serial elisions of the benchmarks. Each running time is the minimum of 10 runs on an Amazon AWS `c4.8xlarge` spot instance, whose technical specifications are given in Figure 10-14.

Overall performance

Figure 10-15 presents the results of our tests. Under each benchmark, the first two rows give the running times of the serial elision under each compilation. These rows should show essentially no difference between Tapir/LLVM and Reference, because for the serial elision of a benchmark, the two pipelines do effectively the same thing. The next two rows gives the work T_1 for each of the two compilations of the parallel code, and the work-efficiency advantage of Tapir/LLVM starts to show. The third pair of rows shows the running times if the parallel code on 18 cores, and here we see some dramatic improvements. For example, Tapir/LLVM is noticeably faster than Reference for AvgFilter, ndMIS, ndST, ndBFS, and detBFS, most of which are codes that run on unstructured graphs. The fourth pair of rows give work efficiency of each compilation. In theory, this value is at most 1.0,[2] because overheads preclude the parallel code to run as fast on 1 processing core as its serial elision. Figure 10-16 shows this pair of rows as a bar graph. Finally, the last pair of rows shows speedup on 18-processing-cores compared to the serial elision. Generally, the numbers favor Tapir/LLVM, validating the efficacy of the Tapir approach.

[2]Occasionally, this value might be slightly larger than 1.0 if the parallel structure of the code admits optimizations that cannot be justified for serial code, resulting in a faster 1-processor execution for the parallel code than the corresponding serial execution. Such is the case for the incST benchmark and several other benchmarks for the Tapir/LLVM compiler.

		Cholesky	FFT	NQueens	QSort	Rectmul	Strassen	AvgFilter
T_S	Ref.	2.549	5.861	2.729	4.305	8.004	6.251	8.868
	Tapir	2.542	5.857	2.733	4.328	8.001	6.216	8.817
T_1	Ref.	4.156	7.383	2.804	5.812	8.391	6.602	9.330
	Tapir	4.012	7.087	2.741	5.372	8.172	6.228	8.859
T_{18}	Ref.	0.375	0.490	0.170	0.572	0.535	1.113	17.113
	Tapir	0.366	0.464	0.165	0.539	0.528	1.112	2.417
T_S/T_1	Ref.	0.613	0.794	0.973	0.741	0.954	0.947	0.950
	Tapir	0.634	0.826	0.997	0.806	0.979	0.998	0.995
T_S/T_{18}	Ref.	6.797	11.961	16.053	7.526	14.961	5.616	0.518
	Tapir	6.945	12.623	16.564	8.030	15.153	5.590	3.648

		Mandel	ndMIS	incMIS	radixSort	SpMV	pRange	kdTree
T_S	Ref.	22.144	7.580	4.223	3.037	1.132	1.974	4.303
	Tapir	22.067	7.609	4.243	3.032	1.136	1.990	4.293
T_1	Ref.	24.625	7.993	4.680	3.085	1.173	2.533	4.453
	Tapir	22.063	7.583	4.296	3.065	1.158	2.420	4.440
T_{18}	Ref.	1.519	10.293	0.463	0.315	0.100	0.448	0.317
	Tapir	1.359	0.550	0.453	0.311	0.099	0.305	0.315
T_S/T_1	Ref.	0.899	0.948	0.902	0.984	0.965	0.779	0.966
	Tapir	1.000	1.003	0.988	0.989	0.981	0.822	0.967
T_S/T_{18}	Ref.	14.580	0.736	9.121	9.641	11.320	4.406	13.574
	Tapir	16.236	13.835	9.366	9.749	11.475	6.525	13.629

		CHull	incST	parallelSF	ndST	ndBFS	detBFS
T_S	Ref.	0.644	3.283	4.003	2.266	2.273	2.753
	Tapir	0.649	3.330	4.030	2.286	2.270	2.750
T_1	Ref.	0.820	3.456	4.236	2.423	2.810	3.280
	Tapir	0.784	3.266	3.996	2.353	2.303	2.736
T_{18}	Ref.	0.094	0.294	0.349	7.310	1.194	2.480
	Tapir	0.093	0.287	0.341	1.663	0.237	0.268
T_S/T_1	Ref.	0.785	0.950	0.945	0.935	0.809	0.839
	Tapir	0.828	1.020	1.009	0.972	0.986	1.005
T_S/T_{18}	Ref.	6.851	11.167	11.470	0.310	1.904	1.110
	Tapir	6.978	11.603	11.818	1.375	9.578	10.261

Figure 10-15: Comparison of the Reference compiler, denoted as "Ref.," to Tapir/LLVM, denoted as "Tapir," across the benchmarks from Figure 10-13. Each measurement is the minimum of 10 runs. The value T_S indicates the running time of the benchmark's serialization, and the values T_1 and T_{18} indicate the running times of the parallel benchmark on 1 and 18 processing cores, respectively. All running times are measured in seconds.

10.8 Related work

This section describes related work in representing parallelism in a compiler IR and analyses and optimizations on parallel programs.

Prior work exploring compiler optimizations focuses on examining the interaction between unstructured parallel threads, in order to remove unnecessary synchronization in

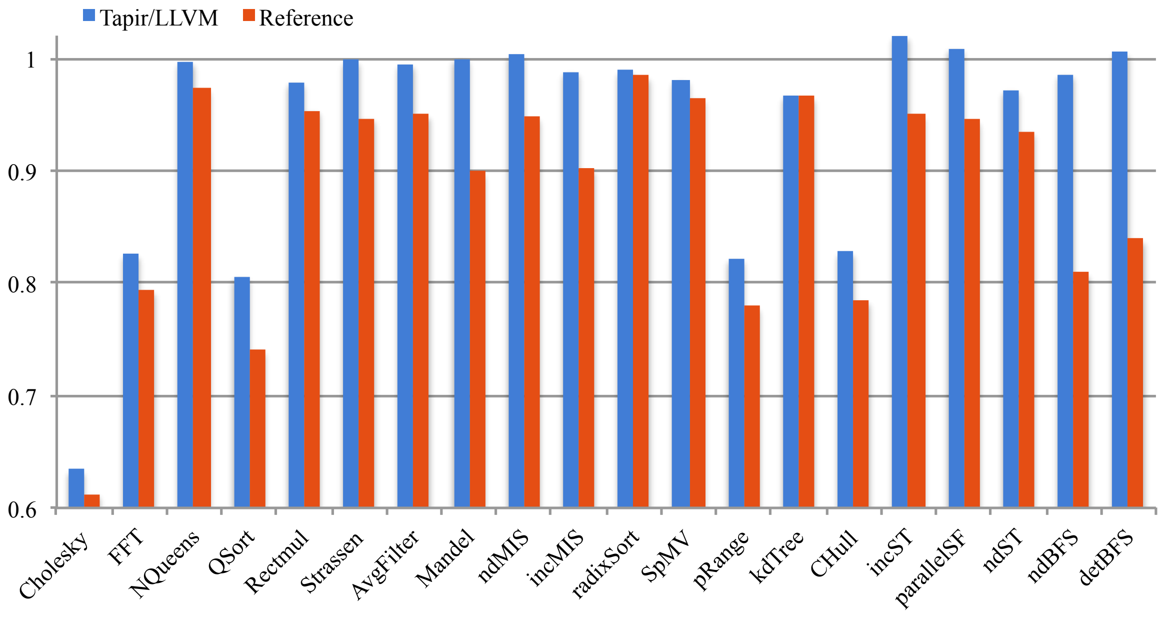


Figure 10-16: The work efficiency of Tapir/LLVM and the Reference compiler. For each compiler, the bar for each benchmark shows the ratio T_S/T_1, where T_s is the running time of the serial elision of the benchmark, and T_1 is the running time of the parallel Cilk code on 1 processor core.

Java programs [10,335] and find references that are not affected by parallel threads and may be safely optimized across parallel control flow [208]. Tapir embeds fork-join parallelism expressed by dynamic multithreading linguistics into the compiler IR.

Compiler optimizations for fork-join parallel programs often evaluate what instructions can happen in parallel [4], based on concurrency mechanisms supported by a particular memory model. Barik, Sarkar, and Zhao [34, 35] use interprocedural analysis to perform various optimizations affecting critical sections of X10 and Habanero Java programs. Tapir embeds logical fork-join parallelism, as distinct from concurrency, into a compiler IR.

Pop and Cohen [316] propose a scheme to translate OpenMP parallel constructs into function calls in the IR that convey their semantics. Although new optimizations can perform optimizations based on these function calls, existing compiler optimizations treat them as opaque, which inhibits these passes from performing most optimizations to avoid creating incorrect code. In contrast, Tapir enables existing compiler optimizations to operate across parallel control flow.

Khaldi *et al.* [217] modify LLVM IR to support OpenSHMEM parallel programs, with the aim of achieving performance in modern network interconnects that support efficient data transfers. Based on SPIRE [216], Khaldi *et al.* augment functions, basic blocks, instructions, identifiers, and types in LLVM IR with execution, synchronization, scheduling, and memory layout information. In contrast, Tapir introduces 3 instructions into LLVM IR to encode fork-join parallelism for shared memory machines, and enables optimizations across parallel control flow.

Chatarasi *et al.* [88] examine polyhedral optimizations on OpenMP programs with serial semantics. Chatarasi *et al.* combine dependency analysis and happens-before analysis in a manner that allows traditional polyhedral optimizers to optimize parallel loops. Tapir embeds fork-join parallelism with serial semantics into LLVM IR to enable general compiler optimizations.

10.9 Conclusion

By embedding the serial semantics of dynamic multithreading programming models directly into the compiler IR, Tapir allows existing compiler optimizations for serial code to effectively optimize comparable parallel codes. As a result, the Tapir approach brings the performance of the dynamic multithreaded codes it supports to be more in line with their serial counterparts, enhancing the degree to which the theoretical performance of these parallel codes is borne out in practice. To conclude, we leave the reader with three interesting considerations regarding the nature of asymmetry in parallelism, the future of parallel optimizations, and extensions of Tapir-like systems to other models of parallel programming.

Reasoning about logically parallel tasks asymmetrically can sometimes simplify the understanding of a parallel program's behavior. When a new task is spawned to execute in parallel with another, it is natural to reason about the logically parallel tasks as symmetric, because their instructions can execute in any relative order. For parallel programs with serial semantics, however, it is always valid to execute the program on a single processor, which asymmetrically executes one parallel task to completion before starting the other. Serial semantics encourages an asymmetric representation of parallel control flow, which is similar enough to its serial elision that most analyses and transformations for the serial elision work on these parallel constructs without much modification.

One of the great benefits of Tapir is that its strategy for representing parallelism makes it easy to write optimization passes specifically for parallel code. Section 10.5 briefly mentioned some parallel optimization passes we implemented, including parallel-loop spawning and unnecessary-synchronization elimination. In addition to helping close the performance gap between serial and parallel versions of code, we hope that the introduction of Tapir will encourage the compiler community to develop and implement many more parallel-optimization passes.

Finally, Tapir allows parallel programs written using a fork-join model of parallelism to benefit from both serial and parallel optimizations. Moving forwards, it is natural to wonder whether other models of parallelism, such as pipeline parallelism [125,239,295] or data-graph computations [264,265,269,298,299,358,361], can take advantage of the Tapir approach.

Chapter 11

Comprehensive Static Instrumentation for Dynamic-Analysis Tools

This chapter presents the CSI compiler instrumentation framework [345]. This work was conducted in collaboration with Damon Doucet, Tyler Denniston, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson.

11.1 Introduction

Key to understanding and improving the behavior of any system is ***visibility*** — the ability to know what is going on inside the system. For application and system software, ***program instrumentation*** — adding special code to monitor the program — has emerged as a popular way for programmers to gain visibility into how their programs are operating. Programmers today can avail themselves of a variety of dynamic-analysis tools, such as race detectors [123, 124, 134, 280, 281, 343, 351], memory checkers [31, 177, 350], cache simulators [121, 377, 407], call-graph generators [171, 205], code-coverage analyzers [390, 397], and performance profilers [171, 329, 401]. These tools generally operate as ***shadow computations*** — executing behind the scenes while the program-under-test runs. Generally, dynamic-analysis tools exploit one of two[1] instrumentation strategies: "binary" instrumentation or "compiler" instrumentation.

Binary instrumentation (e.g., [72, 108, 230, 267, 296, 334, 363, 367]) works by translating the binary executable for a program-under-test into a ***tool-instrumented executable (TIX)*** in which instrumentation code has been inserted by the tool writer using a framework such as Pin [267], Dynamo Rio [72], or Valgrind [296]. Binary instrumentation has proved valuable for gathering detailed information of an executing program, but running-time overheads can be significant, even using "just-in-time (JIT)" techniques. Additionally, the tools are tied to a particular instruction-set architecture. Finally, using a binary-instrumentation framework requires the tool writer to understand some amount about the instruction-set architecture, although binary instrumentation frameworks try to insulate the tool writer as much as possible from the details.

[1]Another strategy is ***asynchronous sampling*** (e.g., [77,171]), which can provide low-overhead solutions for some analytical tools. Asynchronous sampling appears to be ineffectual for many important tools, such as code coverage and memory checking.

Advantage	*Binary*	*Compiler*	*CSI*
Tool works on third-party libraries.	✓	—	—
Tool writers do not need to know compiler internals.	✓	—	✓
Tool writers do not need to know assembly language.	—	✓	✓
Tool is platform independent.	—	✓	✓
Tools can exploit compiler analyses and optimizations.	—	✓	✓
Tools can rely on custom compiler analyses.	—	✓	—

Figure 11-1: A comparison of the advantages of binary-instrumented tools, compiler-instrumented tools, and CSI-tools.

With ***compiler instrumentation*** (e.g., [133, 350, 352]), the tool writer modifies the compiler to insert instrumentation code into the program-under-test, if a command-line switch so indicates. Unlike binary instrumentation, compiler instrumentation can provide detailed runtime information without tying tools to a specific architecture, but at the price of requiring source code for any third-party libraries if those, too, are to be instrumented. Tools can exploit compiler analyses to make the instrumentation more "surgical," and the inserted code itself can be optimized by the compiler. The source code for the compiler, however, can become complex, as different tools demand overlapping, but different, instrumentation. But perhaps the biggest downside of compiler instrumentation is that the development of new tools requires compiler work, which many potential tool writers are ill equipped to do, and thus raises the bar for building new and innovative tools.

Figure 11-1 compares the relative advantages of binary instrumentation and compiler instrumentation, as well as the CSI approach taken in this chapter, which we next discuss. Because no technique dominates any other, each has domains where it has proved more useful than the others, and even mixed approaches (e.g., [129, 180]) have been productive.

The CSI approach

CSI (comprehensive static instrumentation) focuses on compiler instrumentation and, in particular, on making it easier for tool writers to build effective platform-independent tools without doing compiler work themselves. CSI provides an application program interface (API) consisting of functions — ***hooks*** — that are automatically inserted into the compiled code of the program-under-test at every important location, such as function entry and exit, basic-block entry and exit, before and after memory operations, etc.[2] Tool writers can insert their own instrumentation into the program-under-test by simply writing a library that defines the semantics of relevant hooks, and then statically linking their compiled library with the program-under-test to produce a TIX. When the TIX is executed, the program-under-test performs normally while the tool-inserted instrumentation performs shadow computation each time a hook is invoked.

At first glance, this brute-force method of inserting hooks at every salient location in the program-under-test seems to be replete with overheads. For example, the instrumentation of a memory operation incurs the function-call overhead of a hook (potentially two, since we have defined the CSI API to insert one hook before and another hook after each memory operation). If a tool does not use this hook, it provides only running-time overhead.

To overcome these overheads, CSI employs modern compiler features such as ***link-time***

[2]Compare this to traditional compiler instrumentation, in which a tool instruments only the events it requires.

optimization (LTO) [368]. LTO is now readily available in most major compilers, including GCC [140] and LLVM [260]. LLVM with LTO enabled produces a ***bitcode*** file instead of a native object file. We refer to a source file, an object file, or a bitcode file as a ***translation unit***, or ***unit*** for short. A bitcode file encodes the compiler's internal representation of the unit, which allows LLVM to further optimize and transform the unit during the linking stage. Thus, when the bitcode files of the program-under-test are statically linked with bitcode files of a CSI-tool, the instrumented program-under-test can be optimized as a whole. LTO elides unnecessary instrumentation, inlines many of the CSI-tool-defined hooks, and otherwise optimizes the running time of both the program-under-test and the shadow computation.

When the instrumented program-under-test is statically linked, it can be converted to a production executable by linking against the ***null tool***: the CSI-tool consisting entirely of ***null hooks***, where the hook simply returns without looking at its arguments. In this case, LTO is robust enough to elide all the instrumentation, as we shall document in Section 11.5, producing a TIX as efficient as a normally compiled executable. The CSI implementation we have built for LLVM — ***CSI:LLVM*** — defines all hooks to be null using ***weak symbols*** [75, p. 680]. Consequently, if a particular CSI-tool fails to define a hook with a ***strong symbol***, the definition used is the null hook which the compiler automatically elides. The resulting TIX runs as efficiently as if CSI had never inserted the unnecessary instrumentation.

CSI simplifies the rapid development and sharing of new tools. CSI allows tool writers to implement novel tools simply as C libraries, without having to understand any specifics of a mainstream compiler's multimillion-line codebase.[3] A tool writer can furthermore share her new, prototype tool simply as library, without having to convince would-be tool users to download and use her custom compiler or to convince compiler developers to accept her modifications. CSI thus enhances the ability of programmers to examine the dynamic execution of a program in a principled manner, in order to deduce its behavior and performance characteristics.

With traditional compiler instrumentation, the tool user not only has an executable for each tool, he must keep track of which object files have been compiled with which switches. In contrast, CSI-tool users avoid this, because they simply have one instrumented bitcode file for each translation unit and a separate TIX for each CSI-tool.

CSI simplifies the compiler source. Because the number of tools based on compiler instrumentation has been growing, the current LLVM compiler source contains custom instrumentation for many different tools, leading to a plethora of conditionals depending on a host of command-line compiler options. CSI allows for a single option — to instrument or not to instrument — which can significantly simplify the compiler source. The customization for a particular CSI-tool occurs at link time, not at compile time. Section 11.5 shows that LTO can be as efficient as compile-time optimization, and thus the delayed binding of which calls to instrument does not cause performance to suffer.

Design considerations

The current API for CSI is not yet as "comprehensive" as we hope will evolve over time. We prioritized which runtime events to instrument based on need (the demonstration tools described in Section 11.5), but the CSI API is designed to be extensible, and so new instrumentation can be added as needs grow. CSI inserts hooks into the program-under-test

[3]At the time of writing, LLVM's codebase is approximately 3 M lines, and GCC's is approximately 14 M lines.

for entry to and return from functions, the start and end of basic blocks, before and after each function call, and before and after each memory operation. Many more kinds of hooks could be implemented to instrument, e.g., atomic instructions, floating-point instructions, front-end language features such as loop iteration events, etc. We chose a minimal set of hooks that allowed us to build seven example CSI-tools, fully anticipating that the interface will grow to encompass hooks needed by other CSI-tools. We chose to implement a minimal "core" set, because we felt it was best to add new instrumentation on an as-needed basis in order to keep the interface simple.

A potential concern of CSI, compared to traditional compiler instrumentation, is that a CSI-tool cannot rely on tool-specific (custom) compiler analyses to decrease overheads. The CSI API, however, allows the results of compiler analyses to be exported. Analysis results used by many tools for optimization purposes can be exported through the hooks using "properties." A hook invoked on a memory operation, for example, might be called with a "property" specifying that the location is guaranteed to be on the stack. If whether a given "property" holds is known at link time and a tool branches depending on that condition, LTO can constant-fold the test and properly elide the instrumentation if the condition does not hold. Although CSI's "property" mechanism is more than sufficient for the seven CSI-tools we have implemented thus far, as CSI continues to develop, we anticipate using other means, such as auxiliary tables, to export less frequently used analysis results.

The final piece of our design provides a set of front-end data (FED) tables that map each instrumented ***IR object*** (e.g., basic block, function, memory load, etc.) to its location in the source code. CSI assigns all instrumented IR objects unique ID's which a tool writer can track and iterate through.

Although this chapter focuses on how CSI can use LTO to overcome instrumentation overheads, CSI does not, in fact, rely on LTO for correctness or performance. Linking the object file of a CSI-tool with the object files of a program-under-test compiled with CSI suffices to produce a correct TIX, albeit a potentially slow one. Furthermore, CSI provides an additional advanced feature, called ***CSI:CTO***, that allows tool users to link CSI-tool bitcodes with a program-under-test at compile time. CSI:CTO enables ordinary compile-time optimizations to elide unnecessary instrumentation. CSI:CTO thereby allows tool users to avoid the potential build-time overheads of LTO and to use any standard linker to produce an optimized TIX. To simplify the description of CSI, however, this chapter describes CSI's behavior with LTO.

Contributions

This chapter makes the following contributions:

- An API for CSI using compiler-instrumented hooks to build dynamic-analysis tools. The results of compiler analyses are passed to hooks through "properties." Tools can associate instrumentation with source code using CSI's runtime library.
- The implementation of CSI:LLVM, an LLVM implementation of CSI, by modifying the LLVM compiler to insert CSI hooks into programs. When statically linked with the null tool and link-time optimized, programs-under-test run as fast as they do with ordinary compilation.
- A collection of demonstration CSI-tools that explore CSI's utility, ease of programming, and performance. The tools include the null tool, a dynamic call-graph generator, a memory-operations counter, a port of Google's ThreadSanitizer [351], a cache simulator, a lightweight performance profiler, and a code-coverage tool.

Outline

The remainder of this chapter is organized as follows. Section 11.2 presents the CSI API, then Section 11.3 shows an example tool using the API. Section 11.4 overviews the implementation of CSI:LLVM, and Section 11.5 describes seven CSI-tools we built to explore the CSI approach. After reviewing related work in Section 11.6, we offer some concluding remarks in Section 11.7.

11.2 The CSI instrumentation API

This section presents the CSI interface. We have collected a small set of hooks with which a large variety of tools can be implemented. Section 11.5 describes several demonstration tools. Although we have chosen the runtime events to instrument based on need and convenience, the API is designed to be extensible, thus new instrumentation can be easily added as new, unsupported tools come to light.

We have chosen to instrument the ***intermediate representation (IR)*** of programs-under-test, which represents a middle ground between source code (dependent on programming language) and compiled machine code (dependent on target architecture). The IR acts as a simple virtual instruction-set architecture which translates easily to machine code. IR is typically organized into a set of “basic blocks,” where each ***basic block*** is a sequence of instructions with no incoming branches except to its entry point, and no outgoing branches except from its exit point. We have focused on designing and implementing a small, core set of instrumentation, and thus have left instrumentation specific to a front-end or a back-end[4] to future research.

CSI’s instrumentation hooks are organized into five groups: initialization, functions, basic blocks, call sites, and memory accesses. To provide flexibility to tool writers, a hook exists both for just before the event, and just after. For example, one tool may wish to save a memory address’s value before it is overwritten (and thus would use the `__csi_before_store` function), while another may prefer to save the stored value (and thus use `__csi_after_store`). Each hook provides parameters which describe the instrumentation (such as the number of bytes accessed in a memory reference). Importantly, LTO will generally elide any parameters not used by a tool.

Other than hooks for initialization, each hook names an IR object, such as a basic block or a memory operation. CSI gives each such IR object a unique integer identifier within one of (currently) six IR-object categories:

- functions,
- function exits,
- basic blocks,
- call sites,
- loads, and
- stores.

Within each category, the ID’s are consecutively numbered from 0 up to the number of such objects minus 1. The range of ID’s for each category is extended during unit initialization, which happens at the beginning of the program and, in the case of dynamic linking, as new units are loaded in. By maintaining a contiguous set of ID’s, the tool writer can easily track IR objects and iterate through all IR objects in a category.

[4]Here, a front-end describes an application which translates source code to IR (such as LLVM’s Clang), whereas a back-end translates IR to machine-specific code

```c
typedef int64_t csi_id_t;

typedef struct {
  csi_id_t num_func;
  csi_id_t num_func_exit;
  csi_id_t num_bb;
  csi_id_t num_call;
  csi_id_t num_load;
  csi_id_t num_store;
} instrumentation_counts_t;

// Hooks to be defined by tool writer
void __csi_init();

void __csi_unit_init(const char * const file_name,
                     const instrumentation_counts_t * const counts);
```

Figure 11-2: CSI hooks for initialization.

In general, each (non-initialization) CSI hook additionally passes semantic information and "properties" about the IR object it names. Conceptually, the semantic information describes what the named IR object does. For example, the semantic information for a load details what memory location is read and how many bytes are read from that location. Intuitively, CSI hooks aim to pass sufficiently rich semantic information to allow a shadow computation to mirror the operation on its own shadow data structures. CSI also passes each hook an argument `prop`, which is a "property." We describe in detail below the semantic information and properties passed to each hook.

To relate a given IR object to locations in the source code, CSI provides "front-end data (FED)" tables, which provide file name and source lines for each IR object given the object's ID.

The remainder of this section describes CSI's API in detail.

Initialization

CSI provides two initialization hooks, shown in Figure 11-2. The `__csi_init` hook is called when the TIX is run, before both the `main` function and the initialization of global variables.

Because the ordering of global constructors is undefined, tool writers must be careful, because static data used by tools might not yet be initialized during the construction of another global object. As is consistent with good coding style (see, for example, the section on "Static and Global Variables" in the Google style guide [168]), we suggest that tool writers ensure that objects with static storage duration (global variables, static variables, static class member variables, and function static variables) be "plain old data": only `int`'s, `char`'s, `float`'s, pointers, or arrays of plain old data. To be safe, the tool writer should allocate any global constructable objects used by a shadow computation dynamically with `malloc` (and initialize them) in `__csi_init`, and then access them via a global static pointer.

To ensure that `__csi_init` is called before any other constructor, we assign it the highest execution priority in the list of global constructors. However, if the program-under-test also contains a constructor annotated with the highest priority (via the `init_priority` attribute), the execution order of that constructor relative to `__csi_init` is undefined.

In addition to the global initialization hook, CSI also provides the translation-unit initialization hook `__csi_unit_init`. The `file_name` argument provides the name of the source

```
// Hooks to be defined by tool writer
void __csi_func_entry(const csi_id_t func_id, const uint64_t prop);
void __csi_func_exit(const csi_id_t func_exit_id, const csi_id_t func_id, const uint64_t prop);
```

Figure 11-3: CSI hooks for functions.

file corresponding to the translation unit.[5] The hook provides parameters for the number of each instrumentation type in the unit. This allows a tool to prepare any data structures ahead of time (for example, an array with an element for each basic block). This hook is invoked once for every unit that contributes to the TIX. When multiple contribute to the TIX, the tool writer may not assume that the invocations of `__csi_unit_init` are called in any particular order, except that they all occur before `main`. Once again, in the case of a dynamic library compiled with CSI, `__csi_unit_init` is invoked once per translation unit that contributes to the dynamic library at the time that the library loads.

Functions

Figure 11-3 lists the two API hooks for functions, which are instrumented on entry and exit. The hook `__csi_func_entry` is invoked at the beginning of every instrumented function instance after the function has been entered and initialized but before any user code has run — in LLVM terminology, at the first insertion point of the entry block of the function. The `func_id` parameter identifies the function being entered or exited. Correspondingly, the hook `__csi_func_exit` is invoked just before the function returns (normally).[6] Its arguments include both a function-exit ID `func_exit_id` and the function ID `func_id` of the function being exited from. The function-exit ID allows tool writers to distinguish the potentially multiple exits from the same function. It could be argued that `__csi_func_exit` does not technically need the `func_id` parameter, since the tool writer can maintain the correspondence between function entries and exits. We nevertheless decided to include the argument, because it encodes the semantics of the corresponding `return` instruction. Beyond programming convenience, we felt that it would help tool writers to verify their code by checking that each exit matches up to an entry from the same function.

In an early version of the CSI API, the function-entry and function-exit hooks contained a pointer to the function itself to identify the function. Upon reflection, we determined that function pointers were, in fact, a bad way to identify functions. When libraries are dynamically loaded and unloaded, for instance, an aliasing situation can occur, where two different functions share the same function pointer at different times. Thus, we abandoned function pointers in favor of our own function ID's, which CSI guarantees to be unique to a function even in the face of dynamic loading and unloading. If the containing library for a function is loaded and unloaded multiple times, however, the function can end up with different `func_id` values at different times. While this eventuality could be worked around, we felt that in most instances this minor anomaly would not matter much to a tool writer or her users.

Function entry and exit hooks are inserted only for functions within instrumented translation units. External library functions, such as `malloc`, are not instrumented, because the compiler has no access to the source code. Fortunately, ***link-time interpositioning*** [75, Chapter 7.13] provides a workaround to instrument library functions, as we illustrate for

[5] All strings in the CSI API are zero-terminated.
[6] We have not yet defined the API for exceptions.

```c
// Hooks to be defined by tool writer
void __csi_bb_entry(const csi_id_t bb_id, const uint64_t prop);
void __csi_bb_exit(const csi_id_t bb_id, const uint64_t prop);
```

Figure 11-4: CSI basic-block hooks.

```c
// Value representing unknown CSI ID
#define UNKNOWN_CSI_ID ((csi_id_t)-1)

// Hooks to be defined by tool writer
void __csi_before_call(const csi_id_t call_id, const csi_id_t func_id, const uint64_t prop);
void __csi_after_call(const csi_id_t call_id, const csi_id_t func_id, const uint64_t prop);
```

Figure 11-5: CSI hooks for call sites.

`malloc`. The tool writer defines the function `__wrap_malloc`, which she makes call the real `malloc` using the symbol `__real_malloc`. Within `__wrap_malloc`, arbitrary instrumentation can be placed around the `__real_malloc` call. The tool writer now relies on her user to pass the `--wrap malloc` option to the linker, which causes the symbol `malloc` to be replaced by the symbol `__wrap_malloc` and the symbol `__real_malloc` to be resolved as `malloc`. The result is that all calls to `malloc` in the original source, instead of invoking the real `malloc`, now invoke `__wrap_malloc`, which performs any instrumentation and invokes the real `malloc` through `__real_malloc`.

Link-time interpositioning also allows the tool writer to examine the parameters and return values of an instrumented function, which is not passed directly its function-entry or function-exit hooks. Intuitively, passing this information directly to these hooks is problematic because the parameters and return value of a particular function depend on its signature, whereas the function-entry and function-exit hooks must have the same signature for all functions. Therefore, while it might be technically possible to pass this information to these hooks (e.g., by using variadic functions), doing so seems to significantly complicate the API and place substantial burden on the tool writer to parse the information correctly. Link-time interpositioning allows tool writers to insert arbitrary instrumentation for specific functions without the burden manually parsing the function parameters and return type.

Basic blocks

Figure 11-4 shows the two CSI hooks for basic blocks. The hook `__csi_bb_entry` is called when control enters a basic block, and `__csi_bb_exit` is called just before control leaves the basic block. The `bb_id` parameter identifies the entered or exited basic block.

Call sites

Figure 11-5 lists the `__csi_before_call` and `__csi_after_call` hooks that instrument call sites, the places in the code where functions are called. The `call_id` parameter identifies the call site, and the `func_id` parameter identifies the function being called. It is not always possible for CSI to statically produce the ID of the called function, such as when the function is called indirectly through a function pointer or the function called is not instrumented. In these scenarios, the value of `func_id` is `UNKNOWN_CSI_ID`, a special value defined by CSI to represent an unknown function.

A tool writer should be aware that anomalies can occur when intermingling instrumented

```
// Hooks to be defined by tool writer
void __csi_before_load(const csi_id_t load_id, const void *addr,
                       const int32_t num_bytes, const uint64_t prop);
void __csi_after_load(const csi_id_t load_id, const void *addr,
                      const int32_t num_bytes, const uint64_t prop);
void __csi_before_store(const csi_id_t store_id, const void *addr,
                        const int32_t num_bytes, const uint64_t prop);
void __csi_after_store(const csi_id_t store_id, const void *addr,
                       const int32_t num_bytes, const uint64_t prop);
```

Figure 11-6: CSI memory-operation hooks.

and uninstrumented code. For example, if an instrumented function `F` calls an uninstrumented function `G`, which then calls another instrumented function `H`, the before-call hook will be invoked for the call to `G`, but not for the call to `H`. Similarly, the function-entry hook will be invoked for `H` but not for `G`. The tool writer must handle these situations herself if she wishes her tool to support intermingling of instrumented and uninstrumented code.

Memory operations

Figure 11-6 shows the four CSI hooks for memory operations. The hooks `__csi_before_load` and `__csi_after_load` are called before and after memory loads, respectively, and likewise, `__csi_before_store` and `__csi_after_store` are called before and after memory stores. The argument `addr` is the location in memory, and `num_bytes` is the number of bytes loaded or stored.

Properties

The `prop` argument in each non-initialization hook is a ***property***: a 64-bit unsigned integer that CSI uses to export the results of compiler analysis and other information known at compile time. A particular property, such as whether a function is pure, whether a call site is indirect, whether a load is volatile, whether a memory location is guaranteed not to be shared (useful for race detection), etc. is encoded as a bit field in `prop`. Figure 11-7 lists some example properties.

To understand what properties are good for, imagine that a tool writer wishes to build a race detector capable of detecting races on shared variables. If the tool writer were using conventional compiler instrumentation, it would be a simple matter to avoid instrumenting locations that could not possibly be involved in a race — such as a variable declared `const` or a variable on the stack whose address does not escape the frame — or for which a check would be redundant with other checks — such as a load that occurs before a store in the same basic block. The `prop` argument gives the tool writer access to specific compile-time information about the memory operation being instrumented.

For example, the tool writer might write the code in Figure 11-8. As the figure shows, the tool writer can use standard bit-masking operations to check whether different properties hold. In this example, the `CSI_PROP_LOAD_IS_CONST` mask indicates whether the loaded value is `const`, the `CSI_PROP_LOAD_IS_NOT_SHARED` mask indicates whether it is guaranteed not to be shared, and the `CSI_PROP_LOAD_READ_BEFRE_WRITE_IN_BB` mask indicates whether it is followed by a store in the same block. By bitwise-OR'ing together these masks and then bitwise-AND'ing the result with `prop`, the code efficiently checks whether any of these properties hold. As we shall see in Section 11.4, once the hook is inlined, the linker constant-

```c
// Function properties
// The function is constant.
#define CSI_PROP_FUNC_IS_CONST 0x1
// The function is pure.
#define CSI_PROP_FUNC_IS_PURE 0x2

// Basic-block properties
// The basic block is the entry block to the function.
#define CSI_PROP_BB_IS_FUNC_ENTRY 0x1
// The basic block is a loop header.
#define CSI_PROP_BB_IS_LOOP_HEADER 0x2
// The basic block is the exit of a loop.
#define CSI_PROP_BB_IS_LOOP_EXIT 0x4

// Call properties
// The call is indirect.
#define CSI_PROP_CALL_IS_INDIRECT 0x1

// Load properties
// The accessed value is aligned.
#define CSI_PROP_LOAD_IS_ALIGNED 0x1
// The location read is not shared.
#define CSI_PROP_LOAD_IS_VOLATILE 0x2
// The location read is not shared.
#define CSI_PROP_LOAD_IS_NOT_SHARED 0x4
// The load reads a constant value.
#define CSI_PROP_LOAD_IS_CONST 0x8
// The load is a read-before-write on the address in the same basic block.
#define CSI_PROP_LOAD_READ_BEFORE_WRITE_IN_BB 0x10

// Store properties
// The store is aligned.
#define CSI_PROP_STORE_IS_ALIGNED 0x1
// The location read is not shared.
#define CSI_PROP_LOAD_IS_VOLATILE 0x2
// The location written is not shared.
#define CSI_PROP_STORE_IS_NOT_SHARED 0x4
```

Figure 11-7: Example CSI properties.

```c
void __csi_before_load(const csi_id_t load_id, const void *addr,
                       const int64_t num_bytes, const uint64_t prop) {
  if (prop & (CSI_PROP_LOAD_IS_CONST |
              CSI_PROP_LOAD_IS_NOT_SHARED |
              CSI_PROP_LOAD_READ_BEFORE_WRITE_IN_BB))
    return;
  check_for_race_on_load(addr, num_bytes);
}
```

Figure 11-8: How properties might be used by a memory hook in a race detector.

folds and eliminates the conditional, and it elides the instrumentation for locations that satisfy any of the three properties.

We debated using a struct to pass properties, which would make the API more extensible in that we would not be limited to 64 binary properties. But structs raise issues of forward compatibility, since all code needs to be recompiled whenever a version of the API changes the struct definition. In contrast, an integer word allows new properties to be defined by a new version without requiring the recompilation of old code. Of course, we might eventually run out of property bits (although we cannot currently think of real use cases for more than

```
typedef struct {
  char * name;
  char * file_name;
  int32_t line_number;
} source_loc_t;

// Accessors for various CSI FED tables.
// Return NULL when given an invalid ID.
source_loc_t const * __csi_get_func_source_loc(const csi_id_t func_id);
source_loc_t const * __csi_get_func_exit_source_loc(const csi_id_t func_exit_id);
source_loc_t const * __csi_get_bb_source_loc(const csi_id_t bb_id);
source_loc_t const * __csi_get_call_source_loc(const csi_id_t call_id);
source_loc_t const * __csi_get_load_source_loc(const csi_id_t load_id);
source_loc_t const * __csi_get_store_source_loc(const csi_id_t store_id);
```

Figure 11-9: Accessors for the FED tables.

a few properties), but in that eventuality, CSI will have proved itself to be of real utility, and designing a completely new API will be mandated by community interest. Furthermore, as CSI continues to develop, we anticipate using other means, such as auxiliary tables, to export complex and infrequently used analysis results, such as dominator analysis.

Front-end data (FED) tables

The API as discussed up to this point is complete in the sense that functional tools can (and have been) created using it. The utility of these tools, however, is limited without the ability to relate runtime events back to locations in the source code. For example, a race detector could report that a race occurred on a certain data address by an access at a certain code address, but that would require the tool user to manually translate the code address to a location within the source code, presenting a major usability issue for tool writers.

Thus, in addition to the API hooks for instrumentation, CSI also provides a runtime interface to ***front-end data (FED)*** tables written by the compiler that translate the ID parameters to corresponding source-line information. Presently, the only source-line information we provide is file name and line number(s) within that file, but the design is extensible enough to allow more information to be added with ease.

Figure 11-9 presents the FED accessors, as well as the `source_loc_t` structure that these accessors return.[7] We chose to provide accessor functions to the FED tables, rather than allowing direct access to front-end data, so that we can change the representation of any FED table over time without affecting the correctness of any tools.

We considered using the existing DWARF tables [126] in the executable to access source information. The DWARF line table contains the mapping between memory addresses that contain the executable code of a program and the source lines that correspond to these addresses. By providing hooks with a memory address of the instrumentation call, one could report on the source lines. We opted against this strategy for two reasons. First, obtaining the memory location associated with a line of instrumentation turns out to be problematic and is not implemented for most LLVM back ends. Second, it seemed strange for a communication path between the front end and the IR to go through the architecture-dependent back end. In the end LLVM provides enough information during compile-time to insert CSI's FED tables. Inserting and handling front-end specific source information (i.e.,

[7]A careful reader may notice that the line number is signed, which permits an error value of -1 for when the line-number information is not available.

information besides files and line numbers) may be the target of future research.

CSI ID's

CSI makes management of ID values within the instrumentation hooks simple for the tool writer and fast during execution. Key to the design is that the ID values are contiguous, which can greatly simplify a tool, because it allows the tool writer to use a flat structure, such as a one-dimensional array, to track instrumentation points in the program. For example, the tool writer can keep track of all basic blocks by simply allocating an array of length `num_bb`, where the kth entry in the array corresponds to the kth basic block. The tool writer is responsible for extending the array as new units are loaded, but this bookkeeping can be straightforwardly done in `__csi_unit_init`. Fortunately, reallocating this array will likely be dominated, either by startup cost for statically linked units, or by dynamic library loading for units loaded that way.

Associating a unique ID to each hook could have been done in other ways. For example, we considered separating each ID into two components: the translation-unit ID and the ID of the hook within the unit. This design choice might have simplified CSI's handling of dynamically linked code, but it complicates the job of the tool writer. We felt that the unit ID provides little to no benefit during program execution and requires tool writers to maintain more complicated data structures (such as two-level arrays rather than a single, flat array), thus slowing down the tool. We opted to make handling of ID values easier on the tool writer, as well as faster during execution of the program-under-test, by making the ID's contiguous.

Another design alternative we considered and discarded was placing all instrumentation hooks in the same ID space. We felt this strategy provides no benefits over the structured approach in the final CSI design, and it loses the benefit to the tool writer of separate ID spaces, namely, that a tool can quickly iterate over all ID's in a given IR-object category. A tool that wishes to iterate over all instrumentation is likely in the output phase after program execution. It can do so with a nested loop: the outer loop iterates over each instrumentation type, and the inner loop iterates over the ID ranges for those types.

We have considered providing additional information to convey the relationships between the ID's for different IR-object categories. For example, because the functions in a program partition the basic blocks in the program, a set of basic-block ID's[8] are all associated with the same function ID. Although we can compute all of these relationships between IR objects and present them to the tool writer, through, for example, auxiliary tables, there are many such relationships, and we have yet to find a tool that would benefit significantly from this information. To keep the initial API of CSI relatively simple, we have opted not to provide these tables for the time being.

11.3 An example CSI-tool

To illustrate the simplicity and expressiveness of CSI, we present a sample CSI-tool shown in Figure 11-10. This tool measures the maximum depth of nested calls encountered during the execution of a program-under-test using the instrumentation hooks provided by CSI. It then associates this data to the original source code and prints the corresponding stack trace using the FED tables.

[8]In fact, a contiguous range of basic-block ID's.

```
 97 typedef struct {
 98   csi_id_t call_id;
 99   csi_id_t func_id;
100 } callstack_entry_t;
101
102 static callstack_entry_t *callstack = NULL;
103 static callstack_entry_t *maxstack = NULL;
104 static unsigned callstack_depth = 0;
105 static unsigned max_depth = 0;
106
107 void report_and_cleanup() {
108   fprintf(stderr, "Max call stack:\n");
109   for (unsigned i = 0; i < max_depth; i++) {
110     callstack_entry_t entry = maxstack[i];
111     csi_id_t call = entry.call_id;
112     csi_id_t func = entry.func_id;
113     source_loc_t *const call_source_loc = __csi_get_call_source_loc(call);
114     source_loc_t *const func_source_loc = __csi_get_func_source_loc(func);
115     fprintf(stderr, "%s called by (%s:%d)",
116             func_source_loc.name,
117             call_source_loc.file_name,
118             call_source_loc.line_number);
119   }
120   free(callstack);
121   free(maxstack);
122 }
123
124 void __csi_init() {
125   callstack = (callstack_entry_t *)malloc(MAX_STACK_BYTES);
126   maxstack = (callstack_entry_t *)malloc(MAX_STACK_BYTES);
127   atexit(report_and_cleanup);
128 }
129
130 void __csi_before_call(const csi_id_t call_id, const csi_id_t func_id, const uint64_t prop) {
131   callstack[callstack_depth++] = (callstack_entry_t){ call_id, func_id };
132 }
133
134 void __csi_after_call(const csi_id_t call_id, const csi_id_t func_id, const uint64_t prop) {
135   if (callstack_depth > max_depth) {
136     max_depth = callstack_depth;
137     memcpy(maxstack, callstack, callstack_depth * sizeof(callstack_entry_t));
138   }
139   callstack_depth--;
140 }
```

Figure 11-10: The `stack-track` tool illustrates the CSI API. The tool measures and reports the maximum stack depth and corresponding call stack. For simplicity, this tool assumes that the entire program-under-test contains no uninstrumented code and no indirect calls. CSI allows the tool writer to write only the hooks relevant to her tool. Other hooks default to null hooks.

Throughout the shadow computation, the tool maintains a ***shadow stack*** `callstack` (line 102) by instrumenting call sites. The hook `__csi_before_call` is run just before the TIX calls any function. The tool writer has defined `__csi_before_call` to push the ID of the call and the ID of the callee onto its shadow stack (line 131). Just after the TIX returns from the call, `__csi_after_call` is run. The tool writer has defined `__csi_func_exit` to check the stack depth (line 135), conditionally update the maximum depth `max_depth`, and capture a snapshot of the shadow stack (lines 136 and 137) if `max_depth` changes, and finally pop a frame off the shadow stack (line 139),

The initialization code includes calling `atexit` (line 127) with the tool's reporting and

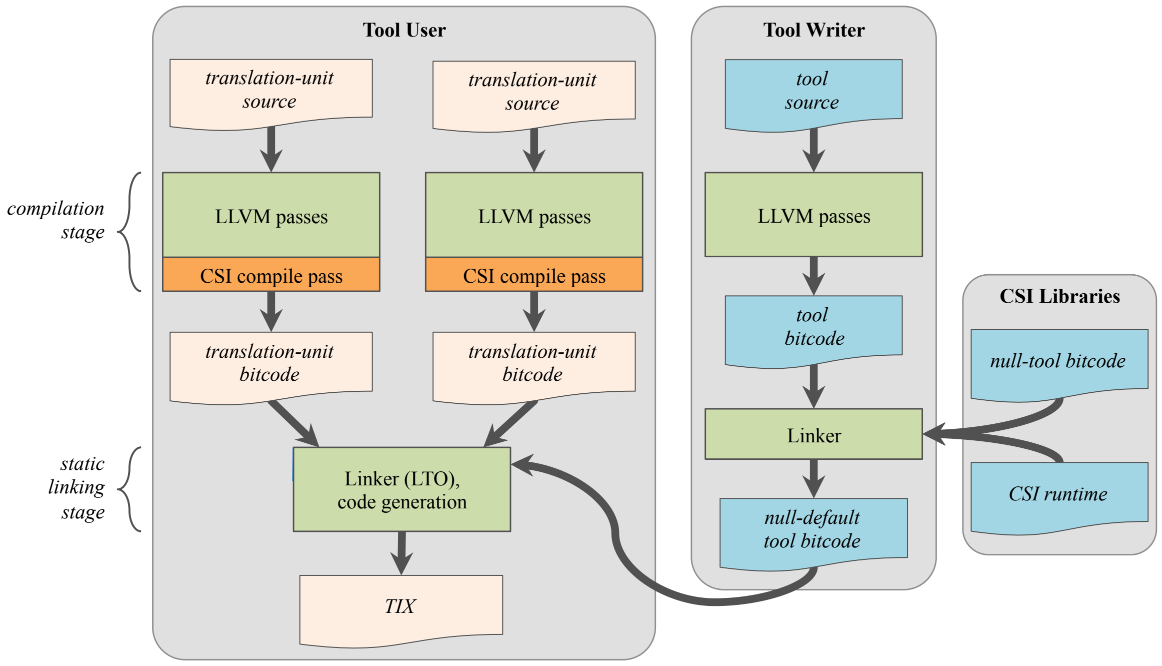


Figure 11-11: The implementation of CSI:LLVM. Gray rounded rectangles distinguish the concerns of the tool user, the tool writer, and the CSI-provided libraries. The tan shapes represent code (in some form) for the program-under-test. The blue shapes represent code for the tool or the CSI system itself. The green boxes are compiler, linker, and code-generation components already provided by LLVM. The orange boxes indicate new LLVM passes that implement CSI.

cleanup function `report_and_cleanup`, which causes this function to be invoked when the TIX terminates. The code for `report_and_cleanup` scans the maximum callstack (lines 109–119) and, for each entry in order, get the front-end data for the call site (lines 113 and 114) and print it out (line 115). Finally, `report_and_cleanup` frees the memory used by the tool (lines 120 and 121).

11.4 Implementation

This section describes the LLVM implementation of CSI. We shall describe the implementation of CSI that interacts with LTO, in order to simplify the description of how CSI allows TIX's to be optimized. We discuss the architecture and several important optimizations that our architecture allows, and then we highlight some limitations.

Architecture

CSI is implemented within the LLVM compiler as two components: a ***CSI compile pass*** and ***CSI runtime***. Broadly, the compile pass inserts calls to the hooks, and the runtime maintains the ID's and aggregates front-end data. Figure 11-11 portrays the entire process of translating program and tool source into a TIX.

Let us first describe how LLVM compiles programs when LTO is enabled. During the ***compilation stage*** of a program, LLVM separately compiles each translation unit into LLVM IR. The compilation stage supports many platform-independent intra-unit (within a single unit) analyses and transformations over the LLVM IR. At the end of this compilation stage, for each compiled unit, LLVM produces a ***bitcode*** file — a compact on-disk binary representation interchangeable with LLVM IR — rather than a native object file. The bitcode files carry all information and metadata produced by the compiler analyses during the compilation stage, which allows further analysis and transformation during the subsequent ***linking stage***, when LTO is invoked.

The CSI compile pass, shown in Figure 11-11, is inserted as an additional compiler pass at the end of the compilation stage, immediately before the generation of the bitcode file. Thus, instrumentation is inserted after all intra-unit compiler optimizations, meaning CSI instruments the *optimized code.* That is normally a good thing, but it does have some consequences. For instance, if a function call is inlined at a particular call site, the `__csi_func_entry` and `__csi_func_exit` hooks will not be instrumented for the function instance invoked at that particular call site.

The CSI compile pass is implemented as an LLVM "module" pass. For each function body, it inserts calls to the appropriate hooks at the designated points within the IR. For properties, the compiler performs any analysis necessary to provide the property argument for the appropriate hooks in the generated IR. Since property arguments are constants, LTO can later constant-fold and propagate to eliminate conditional tests involving these arguments. To assign the hook ID's and create the FED tables, the CSI compile pass creates a set of static global variables within the unit and inserts a call to the runtime's unit initialization function, `__csirt_unit_init`, with pointers to those variables. The CSI pass prepends the `__csirt_unit_init` function to the unit's global constructor list, thereby ensuring unit initialization occurs before execution of the TIX's `main` function or any other global constructors.

Figure 11-12 shows `__csirt_unit_init`, the runtime's unit initialization function. This function works in concert with the CSI compile pass to set the ID values for all of the hooks and to construct the FED tables. This function also calls the tool's unit initialization hook. To understand this function in detail, let us examine how CSI assigns ID's and generates FED tables.

Assigning ID values and building FED tables

Let us first examine how CSI assigns contiguous ID values to hooks. To illustrate the process, consider compiling a unit with m basic blocks. The CSI compile pass numbers the basic blocks in the unit with a ***local*** ID value from 0 through $m-1$. It also creates a static global variable, `bb_base`, inside the unit, which it passes by reference to the unit's call to `__csirt_unit_init`. As Figure 11-12 shows, the CSI runtime maintains a running count of the total number of basic blocks in all loaded units, (line 147). When this unit is initialized, suppose that the units initialized so far have n total basic blocks among them. Then `__csirt_unit_init` sets the `bb_base` of the unit to n, its current running total (line 159), and updates its total basic-block count to $n+m$ (line 160). The ID for each basic block hook is the result of adding its local ID value to the value of `bb_base` read when the hook executes, which produces the values $n, n+1, \ldots, n+m-1$ for this unit.

Function ID's pose an additional complication to this scheme, in order to support passing them to the hooks for calls. In particular, a call hook (for a direct call) takes the ID of its callee, which might be defined in another module. If unit A contains a call to a function and is initialized before the unit B that contains the callee, then the ID of the callee is not known when A is initialized.

To resolve this issue, the CSI compile pass generates a global, weak ***function-ID symbol***

```
141 typedef struct {
142   source_loc_t *func;
143   source_loc_t *bb;
144   // ...Other FED tables...
145 } fed_tables_t;
146
147 instrumentation_counts_t total_counts;
148 fed_tables_t global_fed_tables;
149
150 void __csirt_unit_init(const char * const name,
151                        const instrumentation_counts_t *unit_base_ids,
152                        const fed_tables_t * const fed_tables,
153                        const instrumentation_counts_t * const counts,
154                        void (*set_func_ids)(void)) {
155   // Set the bases of the ID categories.
156   unit_base_ids->func_base = total_counts.num_func;
157   total_counts.num_func += counts->num_func;
158
159   unit_base_ids->bb_base = total_counts.num_bb;
160   total_counts.num_bb += counts->num_bb;
161   // ...
162
163   // Set the function ID's.
164   set_func_ids();
165
166   // Construct the FED tables.
167   global_fed_tables.func = (source_loc_t *)realloc(
168       global_fed_tables.func, sizeof(source_loc_t) * total_counts.num_func);
169   for (csi_id_t i = 0; i < counts->num_func; ++i)
170     global_fed_tables.func[i + unit_base_ids->func_base] = fed_tables.func[i];
171   // ...
172
173   __csi_unit_init(name, counts);
174 }
```

Figure 11-12: CSI runtime's unit initialization hook.

for every function defined or called in the unit, which is initialized to `CSI_UNKNOWN_ID`. The call hooks then simply load the value of the callee's function-ID symbol at runtime. The CSI compile pass also generates a function, `set_func_ids`, for the unit, which sets the values of the function-ID symbols for the functions defined in the unit. After `__csirt_unit_init` sets the `func_base` for the unit, it calls back to this generated `set_func_ids` function in the unit (line 164). Although multiple units might create function-ID symbols for the same function, the weakness of these symbols ensures that only one function-ID symbols for each function will survive. Furthermore the value of a function-ID symbol is only set when the unit that defines the corresponding is initialized. Finally, the initialization of these symbols ensures that call hooks will load the value `CSI_UNKNOWN_ID` as the ID of a function defined in an uninstrumented unit.

To construct the FED tables, for each IR-object category, the CSI compile pass collects front-end data for IR objects in that category into a static global array for the unit. This array is passed to `__csirt_unit_init`, which copies the contents of the array onto the end of a global FED table array, as lines 167–170 illustrate.

By using a runtime pass which sets ID values and FED tables when units are initialized, CSI assigns ID values and FED tables correctly, regardless of whether units are statically or dynamically linked.

(a)

```
void func(int i) {
  global += i;
}
```

(b)

```
define void @_Z3funci(i32 %i) #0 {
  call void @__csi_func_entry(...)
  call void @__csi_bb_entry(...)
  call void @__csi_before_load(...)
  %1 = load i64, i64* @global, align 8
  call void @__csi_after_load(...)
  %2 = sext i32 %i to i64
  %3 = add i64 %2, %1
  call void @__csi_before_store(...)
  store i64 %3, i64* @global, align 8
  call void @__csi_after_store(...)
  call void @__csi_bb_exit(...)
  call void @__csi_func_exit(...)
  ret void
}
```

(c)

```
define void @_Z3funci(i32 %i) #0 {
  call void @__csi_func_entry(...)
  %1 = sext i32 %i to i64
  %2 = load i64, i64* @global, align 8
  %3 = add i64 %2, %1
  store i64 %3, i64* @global, align 8
  call void @__csi_func_exit(...)
  ret void
}
```

Figure 11-13: An example of null-hook elimination. **(a)** The original function. **(b)** The compiled IR, where for clarity, IR instructions that set up the arguments for the various hook invocations, as well as the arguments themselves, are not shown. **(c)** The optimized IR after linking with a tool that only implements `__csi_func_entry` and `__csi_func_exit`. LTO elides all the null hooks.

Null-hook elimination

To ensure ***null-hook elimination*** occurs, meaning that hooks unimplemented by the tool writer are properly defined, yet optimized away, the tool writer first links the bitcode files of her tool with the null tool, and then with the program-under-test. The default implementation of all hooks in the null tool is simply an empty function, exported as a weak symbol. By default, the implemented hooks in the tool writer's tool are defined as strong symbols, which override the same weak symbols in the null tool during linking. Thus, any hooks not implemented by the tool writer's tool use the default definitions from the null tool, all of which are simply null hooks. Since LTO optimization passes automatically elide empty functions as part of a dead-code elimination pass, when the definitions of the hooks become available to the link-time optimizer, calls to null hooks are trivially marked as dead code. Consequently, all function calls to the null hooks are removed from the final executable.

As an example of null-hook elimination, consider the source code in Figure 11-13(a), a function that writes to a global variable. For this function, CSI:LLVM produces the IR in Figure 11-13(b). Suppose that this IR is linked with a tool that only implements `__csi_func_entry` and `__csi_func_exit`. LTO will produce the optimized IR in Figure 11-13(c), because calls to hooks not implemented by the tool are eliminated by LTO.

Care must be taken to ensure that a CSI-tool written with to one version of the API is not erroneously linked with an incompatible bitcode file compiled with a different version of CSI:LLVM. Library versioning is a common headache and has provoked solutions such as each library providing version numbers in their header files and at runtime, as support from the automake tools [314], conventions such as Apache's version numbering scheme [20], using namespaces to handle versions, providing a variable whose name encodes the version forcing a link-time error on mismatching, and simply changing the name of the library for different versions. We intend to use a mix of these techniques eventually, but the current implementation of CSI:LLVM punts on this problem, which we rationalize by the APR versioning policy [20], which states that any library that has not reached 1.0.0 does not need to worry about versioning.

Limitations

CSI:LLVM's current approach to instrumentation does contain some limitations, however. For example, the decision to instrument the IR and not the front or back end might lead to a misunderstanding of results gathered by a CSI tool. For example, loads and stores created during machine code generation (which occurs after IR generation) are not instrumented. This scenario occurs during register allocation whenever a register must be spilled to the stack using load and store machine instructions. These memory operations cannot be instrumented in the current implementation of CSI.

Another limitation occurs when instrumenting programs with dynamic libraries. Dynamic libraries must be compiled as ***position independent code (PIC)*** [75][Chp. 7.12], and as the compiler cannot predict runtime addresses within the library, it must invoke tool-provided hooks as PIC function calls. Although not a functional deficiency, we have observed performance issues in this situation. We are investigating why, in these cases, LTO can sometimes fail to perform optimizations to eliminate null hooks or dead code within the hooks. To be conservative and avoid these penalties, libraries should be statically linked with the TIX. Section 11.5 compares runtime overheads incurred by an instrumented library when linked dynamically versus statically.

11.5 Demonstration CSI-tools

This section describes seven example CSI-tools that we built to investigate the properties of the CSI:LLVM implementation:

- **CSI-null**: the null tool,
- **CSI-cgg**: a dynamic call-graph generator,
- **CSI-memop**: a memory-operations counter,
- **CSI-TSan**: a port of Google's ThreadSanitizer [352] race-detection tool,
- **CSI-prof**: a lightweight performance profiler,
- **CSI-reuser**: a cache simulator,
- **CSI-cov**: a code-coverage tool.

Figure 11-14 summarizes the specifications of the benchmarking machine used for all of the experiments.

CPU	Intel Xeon E5-2695 v2
Clock	2.4 GHz
Hyperthreading	Enabled
Turbo Boost	Disabled
Cores per processor chip	12
Processor chips (sockets)	2
L1 data cache/core	32 KiB
L2 cache/core	256 KiB
L3 cache/socket	30 MiB
DRAM	128 GiB DDR3
Compiler	Clang 3.8.0
Operating system	Linux kernel 3.13.0

Figure 11-14: Technical specifications of the machine used for benchmarking. We disabled Turbo Boost to enhance the reliability of time measurements.

	Running time		*Requests/second*		*Slowdown*	
Configuration	min	max	min	max	min	max
Baseline	16.5	16.9	17,755	18,117	—	—
CSI	16.6	16.7	17,946	18,062	0.98	1.01

Figure 11-15: The null tool CSI-null exhibits negligible running-time overheads on the Apache server when compiled with static linking. The measurements show the fastest and slowest of three runs. Running time is measured in seconds, and slowdown is computed based on running time.

The null tool

The null tool CSI-null demonstrates that if a CSI-tool does not use a particular part of the CSI instrumentation, then the running-time cost is essentially zero and the build-time overhead is reasonable. To measure the running-time overhead of CSI-null, we compiled the Apache HTTP server (version 2.4.17) to build with and without CSI-null. We chose Apache as a representative real-world program of moderate size. The SLOCcount tool [405] reports 264 883 source lines of C code. We used the Apache benchmark harness from the ThreadSanitizer repository [167] to issue 300 000 connections with a concurrency level of 20 (meaning up to 20 simultaneous requests may be issued). The benchmark harness then reports a total running time and a mean measurement of the number of requests handled per second.

Figure 11-15 reports the essentially zero running-time overhead of CSI-null for a statically compiled Apache server. By default, the Apache server employs many dynamically loaded libraries. As mentioned in Section 11.4, instrumentation of dynamically linked objects can suffer in performance. Figure 11-16 measures the overhead of CSI-null when Apache is configured to use dynamic libraries instead of statically linked ones. As can be seen in the figure, the running-time overhead is 33 % instead of 0 in this case. Even with dynamic tool-instrumented libraries, CSI:LLVM optimizes each library separately against the null tool in this experiment.

One possible concern that tool users might have regarding CSI:LLVM is that build time might be increased unduly. Figure 11-17 summarizes the results of compiling the Apache server with and without LTO and CSI. When CSI is enabled, we used CSI-null. We report serial build times as well as parallelized build times with `make -j24`. When using 24 parallel jobs, compiling Apache with CSI adds ~14.9 s to the elapsed build time, or 39 % overhead. Using 1 job, CSI increases build time by 14 %. To understand how CSI scales with program

	Running time		*Requests/second*		*Slowdown*	
Configuration	min	max	min	max	min	max
Baseline	15.6	15.9	18,869	19,265	—	—
CSI	20.7	20.8	14,408	14,473	1.30	1.33

Figure 11-16: The null tool CSI-null exhibits nontrivial running-time overheads on the Apache server when compiled with dynamic linking. The measurements show the fastest and slowest of three runs. Running time is measured in seconds, and slowdown is computed based on running time.

# Jobs	*LTO*	*CSI*	*User (s)*	*System (s)*	*Elapsed (s)*
1	—	—	138.3	31.7	177.7
1	✓	—	138.4	31.7	181.2
1	✓	✓	164.2	31.7	203.0
24	—	—	134.4	30.6	38.2
24	✓	—	132.8	29.9	43.3
24	✓	✓	158.6	29.7	53.1

Figure 11-17: Compile-time overhead of LTO and CSI when building Apache HTTPD. The first three lines show the running times when `make` is run serially: with the default `make`, running the default with LTO, and running the default with LTO and CSI. The last three lines show the running times for 24 jobs. We measured the user, system, and elapsed time from `/usr/bin/time`.

# Jobs	*Instrumentation*	*User (s)*	*System (s)*	*Elapsed (s)*
1	CSI-TSan	178.1	34.5	227.8
1	TSan	145.0	33.0	192.8
24	CSI-TSan	172.9	32.1	60.5
24	TSan	140.4	31.1	45.8

Figure 11-18: Compile-time overhead of a port (CSI-TSan) of the ThreadSanitizer race detector to CSI compared to the original ThreadSanitizer (TSan), measured when building Apache HTTPD.

size, we compiled Apache both with the original ThreadSanitizer race detector [352] and with CSI-TSan, which will be described shortly. Figure 11-18 shows the results: compiling with CSI adds 32 % build-time overhead over the original ThreadSanitizer when the tool is built using 24 jobs and 18 % when it is built using a single job.

Dynamic call-graph generator

Our second demonstration CSI-tool provides an opportunity to compare performance of the CSI approach with binary instrumentation, and in particular, with a Pintool [267]. We implemented a dynamic call-graph generator in CSI, called CSI-cgg, which was inspired by the dynamic call-graph generator Pintool from the Pin documentation [194].

We measured running-time overhead using the Apache HTTP server compiled with and without CSI-cgg. Because the Apache server configuration is multithreaded, we adapted the original Pintool to be thread-safe, and we built CSI-cgg as the same tool using CSI. Because the CSI:LLVM infrastructure does nothing special for multithreading, CSI-cgg must implement multithreaded data structures. In particular, both CSI-cgg and the Pintool construct a thread-local call graph to avoid unnecessary locking at runtime. The thread-local call graphs are merged together and printed to a file when the application exits. We used the GLib [161] binary search tree and linked list.

Configuration	*Running time (s)*	*Requests/second*	*Running-time slowdown*
Baseline	15.4	19,454.6	—
Pintool	312.1	961.3	20.3
CSI-tool	113.9	2,633.2	7.4

Figure 11-19: Comparison of the CSI-cgg dynamic call-graph generator and the equivalent Pintool on the Apache server compiled with dynamic libraries.

```
static long accesses_by_size[4];

void __csi_init(const char * const name) {
    atexit(print_results);
}

void __csi_before_load(const uint64_t load_id,
                       const void *addr,
                       const int num_bytes,
                       const uint64_t prop) {
    accesses_by_size[__builtin_ctz(num_bytes)]++;
}

void __csi_before_store(const uint64_t store_id,
                        const void *addr,
                        const int num_bytes,
                        const uint64_t prop) {
    accesses_by_size[__builtin_ctz(num_bytes)]++;
}
```

Figure 11-20: The CSI-memop tool counts memory operations.

For CSI, we compiled the server daemon with CSI-cgg. For Pin, we started the daemon under Pin control with an equivalent tool. The Pintool was configured not to instrument system libraries to ensure as fair a comparison as possible. Figure 11-19 summarizes our findings. Whereas use of the Pintool resulted in a server slowdown of 20.3 $\times$, CSI-cgg resulted in only a 7.4 $\times$ slowdown.

Figure 11-19 presents running-time overhead measurements of both the CSI-cgg call-graph generator and the equivalent Pintool in the same Apache benchmarking configuration. These experiments used a dynamically linked Apache server (in which each of Apache's dynamic libraries is tool instrumented). CSI-cgg was approximately 2.7 $\times$ faster than the equivalent Pintool in running time.

Memory-operations counter

The third CSI-tool, called CSI-memop, counts the number of loads and stores in a program by incrementing a global counter for each memory operation. Counting memory operations presents a great opportunity for optimization. The Pin documentation [194] describes how to make an instruction-counting tool more efficient by counting the number of instructions once for each basic block and then augmenting the global counter with that sum once every time the basic block is executed, rather than incrementing once per instruction. To count memory operations, the same idea can be used, but a Pintool writer must implement this optimization herself. In contrast, the writer of a CSI-tool can write the natural increment-for-each-memory-operation code, as shown in Figure 11-20, which CSI:LLVM optimizes automatically.

We built a microbenchmark to evaluate CSI-memop and the optimized Pintool. The microbenchmark consists of a billion iterations of a loop that reads several variables of different types, increments them, and stores them back. CSI:LLVM was able to infer the loop bounds, causing the instrumentation to be lifted out of the inner loop. The net result was that the CSI-memop incurred essentially zero overhead compared to the baseline program. In comparison, the Pintool exhibited measurable overhead for as few as four variables. The goal of this experiment was not to show that CSI-memop was so many percent faster, but rather that it leverages the inherent optimization capability of CSI:LLVM.

ThreadSanitizer

Our fourth tool, CSI-TSan, is a port of ThreadSanitizer [352], which performs race detection by instrumenting every load and store and intercepting calls to `pthread` function calls, such as accesses to condition variables [148]. The ThreadSanitizer tool provides an excellent opportunity to compare the CSI approach with traditional compiler instrumentation. Our studies with CSI-TSan demonstrate the following:

- An existing compiler-instrumentation tool, such as ThreadSanitizer, can be ported to CSI remarkably easily.
- The ported CSI-TSan tool produces code that runs only marginally slower than the original ThreadSanitizer tool.
- CSI properties can efficiently export compiler analyses to the CSI hooks at runtime.
- CSI can support a large tool compared to the other six example tools, all of which are fairly small.

The standard ThreadSanitizer implementation has two parts: a compiler pass and a runtime library. The compiler pass includes tool-specific optimizations as well as code to instrument every load and store. One such tool-specific optimization is to omit instrumentation of a load if there is a store to the same address in the same basic block.

CSI's instrumentation interface differs slightly from that of ThreadSanitizer. For example, ThreadSanitizer has a separate load instruction for each operand size (`__tsan_read1` for 1-byte reads, `__tsan_read2` for 2-byte reads, etc.), whereas CSI-TSan employs a single function that takes an address and size. It turns out that underneath the width-specific instructions, ThreadSanitizer employs a single function that takes an address and a size, and CSI-TSan simply translates the API calls and invokes the original library functions, requiring a total of 15 lines of code.

Of the 188 ThreadSanitizer regression tests, CSI-TSan passes all but 20. Those 20 tests require the following features that CSI:LLVM does not currently support:

- ThreadSanitizer-specific annotations to help race detection, such as "x happens before y,"
- atomic instructions and variables,
- Virtual pointers and C++ objects, and
- Races involving static variables initialized in functions.

Figure 11-21 presents a running-time-overhead comparison of ThreadSanitizer and CSI-TSan with and without using CSI properties. CSI-TSan employs the "read before write" property on the `__csi_before_load` hook. For a race detector, loads that occur before stores in the same basic block need not be instrumented. CSI-TSan checks the value of this property and only instruments loads for which it is false. The LTO process then can elide any calls to the `__csi_before_load` hook whenever it can prove that the load is subsumed by a write in the same basic block. This optimization reduces minimum overhead on the

	Running time (s)		*Overhead*	
Configuration	min	max	min	max
Baseline	5.33	9.48	—	—
ThreadSanitizer	13.64	21.12	1.43	3.96
CSI-TSan, no properties	28.02	43.73	2.95	8.20
CSI-TSan with properties	19.80	33.65	2.08	6.31

Figure 11-21: Comparison of running-time overheads for ThreadSanitizer and CSI-TSan without and with using CSI properties.

running time of CSI-TSan from 295 % to 208 %.

Performance profiler

For our fifth example CSI-tool, we implemented a simple, lightweight function-based profiler, called CSI-prof, which uses the read-timestamp-counter instruction `rdtsc` to measure the time spent in the program. At each function entry and exit, the time since the last measurement is computed and added into a global variable. A real profiler would do the bookkeeping to assign those times to a function, but for this tool, we simply summed up all the time measurements. CSI-prof demonstrates that CSI can do more than instrument code for correctness: it can also measure performance.

We measured a classical doubly recursive Fibonacci program. We coarsened the base case of the recursion so that we could vary the number of instructions compared to the number of function calls. Given that the processors on our machine are clocked at 2.4 GHz and that the run-to-run variance was less than 1 %, we calculated that the `rdtsc` instruction was taking about 22.2 ns. The running time of the program could then be modeled as $T = 22.2\,\text{ns} \cdot C + N/2.4\,\text{GHz}$, with a relative root-mean-square error of less than 0.7 %.

A comparable tool is the profiler gprof [171], which employs asynchronous sampling, interrupting the program periodically and looking at the call stack to record in which function the program counter lies. In comparison, CSI-prof measures every single function call. Not surprisingly, gprof suffers less overhead than CSI-prof. For example, for Fibonacci with no coarsening, gprof slows down the program by about a factor of 5, and CSI slows it down by another factor of 2. Sampling has the disadvantage, however, that it is inaccurate for short runs. For example, some of our runs took only 0.36 seconds, and gprof's measurements varied by more than a factor of 2 from run to run, whereas CSI-prof produced the same answer to within 1 %.

Cache simulator

The sixth example CSI-tool is a cache simulator, called CSI-reuser, based on reuse-distance analysis [121, 377]. Reuse distance is essentially a plot of the cache-miss rate of a program as a function of its cache size. To calculate reuse distances, CSI-reuser instruments every load and store using a data structure similar to that of [121], which can calculate reuse distance in $O(n \log n)$ time, where n is the total number of memory operations.

Other cache simulators, such as Cachegrind [296] and Loca [120, 412], employ binary instrumentation. Cachegrind instruments every load and store, even in uninstrumented libraries. The Pintool Loca employs a sampling strategy to reduce overhead. We found Loca to be sensitive to the version of the kernel and of other system utilities, and we have

```
221 csi_id_t num_basic_blocks = 0;
222 bool *block_was_executed = NULL;
223
224 void report() {
225   fprintf(stderr, "Basic blocks not executed:\n");
226   for (csi_id_t i = 0; i < num_basic_blocks; i++) {
227     if (!block_was_executed[i]) {
228       if (source_loc_t *const source_loc = __csi_bb_get_source_loc(i))
229         fprintf(stderr, "%s:%d\n",
230                 source_loc.filename,
231                 source_loc.line);
232     }
233   }
234   free(block_was_executed);
235 }
236
237 void __csi_init(const char *const name) {
238   atexit(report);
239 }
240
241 void __csi_unit_init(const char * const name,
242                      const instrumentation_counts_t * const counts) {
243   num_basic_blocks += counts->num_bb;
244   block_was_executed = (bool *)realloc(block_was_executed,
245                                        num_basic_blocks * sizeof(bool));
246 }
247
248 void __csi_bb_entry(const csi_id_t bb_id) {
249   block_was_executed[bb_id] = true;
250 }
```

Figure 11-22: The CSI-cov code-coverage tool, which reports lines of code that are not executed.

as yet been unable to get Loca to work because of incompatibilities with our system. We hope to resolve those problems and compare Loca to CSI-reuser in future work.

Code coverage

Our seventh and final CSI-tool, called CSI-cov, illustrates the advantages of having contiguous values for basic-block ID's. CSI-cov tracks which basic blocks are executed over the course of the program, reporting on exit the source-line data for basic blocks that were never executed. The tool, which is shown in Figure 11-22, is simple. In 30 lines, the CSI framework enables this useful compiler-based tool to be implemented as a library. Without the CSI framework, this code-coverage tool would require considerably more development effort, as well as an understanding of the internals of the compiler.

The code-coverage tool tracks which blocks were executed through the global array `block_was_executed` by implementing the initialization hooks and `__csi_bb_entry`. The `__csi_init` hook simply registers the output routine to be called when the program exits. The `__csi_unit_init` hook extends the array to include the new basic blocks added by this unit. The `__csi_bb_entry` hook marks when a basic block is executed.

At the end of the computation, the `report` function runs through all basic blocks, and if a given basic block was not executed, it uses the basic-block FED table to report the source location of the basic block to the user. The contiguity of `bb_id` values simplifies the computation. The `for` loop in lines 226–233 iterates through all the basic blocks straightforwardly, even if basic blocks come from different units.

11.6 Related work

The CSI framework provides a standard collection of instrumentation hooks so that a tool writer can develop compiler-based dynamic-analysis tools without needing to modify the compiler. In this section, we survey the literature to provide context for CSI.

Complementing the compiler-instrumentation approach have been many general frameworks developed to allow tools to rewrite or instrument arbitrary binaries in order to perform dynamic analysis [72, 108, 230, 267, 296, 334, 363, 367]. Unlike CSI, a tool using binary instrumentation can instrument any executable without access to the source code. The main downside is that most binary-instrumentation frameworks are platform specific, making the tool platform dependent. A few notable exceptions include EEL [230], FIT [108], and Valgrind [296], which target multiple back-ends.

Many frameworks have been developed for bytecode instrumentation [47, 70, 73, 236, 270, 394, 406]. These frameworks are tied to the Java Virtual Machine (JVM), even though the JVM itself is designed to be platform independent.

Many dynamic-analysis tools have been developed using compiler instrumentation, such as [134,149,171,206,240,281,346,350,351,415], where the tool modifies the compiler to insert tool-specific instrumentation tailored to its needs. Few general compiler-instrumentation frameworks extend beyond tool-specific instrumentation. Two exceptions are SASSI [373], a low-level assembly-language instrumentation tool for GPU's, and TAU [354], which focuses on instrumenting high-level C++ language features. Both exhibit a different focus from CSI, requiring users to insert source commands to instruct the compiler what to instrument.

The lack of a general compiler-instrumentation framework should not be surprising, since LTO has only recently been widely available in mainstream compilers. Without the support of LTO or comparable compiler technology, tools would be ridden with overhead, but CSI demonstrates that a general compiler-instrumentation framework is viable today, providing tool writers with a new avenue for instrumentation.

11.7 Conclusion

The CSI framework supports the rapid development of high-performing compiler-based dynamic-analysis software tools, granting performance engineers insight into a program's behavior. We have implemented the framework in the LLVM compiler and demonstrated its efficacy through demonstration tools. We are optimistic that the framework — or at least the concepts of CSI — will be adopted by the community to assist in the development of a wide range of tools. We hope that tool writers will give us feedback on how best to enrich CSI with additional features while keeping the API simple and minimal. We plan to release CSI:LLVM publicly under a liberal open-source license so that the community can help evolve the instrumentation framework. To this end, we have initiated the public code review process to upstream of CSI implementation into the main branch of LLVM [111, 112, 258].

Chapter 12

Life after Moore's Law

To conclude this thesis, let us return to the discussion in Chapter 1 on the end of Moore's Law and what it implies about the need to engineer software for performance. Mounting evidence indicates that Moore's Law will end within the next 5 years. The final *International Technology Roadmap for Semiconductors* [203] states that, "By 2020–25... it will become practically impossible to reduce [transistor sizes] any further." Meanwhile Intel, the leader in semiconductor-device technology, has stated that they plan to use their older 14 nm and next-generation 10 nm transistor technologies for longer amounts of time [201, p. 14], indicating a clear slowing from the historical Moore's Law rate of doubling transistor densities on an integrated circuit every two years.

Could other sources provide rapid growth in computer performance after Moore's Law ends? To tackle this question, I worked with a team of MIT researchers with broad expertise in computing technology and economics, including Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Charles E. Leiserson, Daniel Sanchez, and Neil C. Thompson. After examining a host of promising technologies, we have concluded that the answer is yes, because there is plenty of opportunities to increase computer performance [247]. In particular, we find these opportunities in the higher levels of the ***computer stack***: the layers of technology built upon transistors and other semiconductor devices that deliver useful applications. These layers of technology organize the transistors at the "bottom" into circuits that perform calculations (hardware architecture), coordinate those calculations as efficient problem-solving routines (algorithms), and combine those routines into functionality available to the user (software). As semiconductor physics and silicon-fabrication technologies at the bottom of the computer stack cease delivering performance gains, the top of the stack will offer opportunities to drive up computer performance through ***performance engineering***: restructuring a computation either to reduce the number of operations needed to solve a problem or to perform the operations more quickly.

Whereas Moore's Law has driven up performance on a predictable schedule, performance engineering will produce opportunistic, uneven, and sporadic gains, typically improving just one aspect of a particular computation at a given time rather than "lifting all boats" as Moore's Law has done. Moreover, Moore's Law improvements have been largely *invisible* to higher levels of the computer stack, meaning that other parts of the system don't need to adapt to obtain the advantages provided by those improvements. In contrast — and unfortunately — performance-engineering changes are often *visible*, meaning that they may require other parts of the system to change in order to exploit, or even tolerate, those changes. When performance-engineering modifications percolate through a system, massive engineering effort can be required to correctly implement and test the changes. How can programmers contend with the complexity of performance-engineering and visible changes in the post-Moore era?

Although some software technologies have historically found success in providing software efficiency while sheltering programmers from performance concerns, these technologies seem limited in their ability to address the software performance concerns of the post-Moore era. Software performance engineers have developed high-performance libraries, such as the Intel Math Kernel Library [200], to provide fast implementations of some commonly used routines. For an arbitrary piece of code, however, programmers cannot always rely on an optimized library being available. Meanwhile, compiler technology has been successful in encapsulating some software optimization tasks, such as register allocation and modifications to program control flow. In the 1970's and 1980's, for example, optimizing compilers allowed programmers to use high-level control-flow constructs in place of `goto` statements [410]. As the matrix-multiplication case study presented in Chapter 1 illustrates, however, optimizing compilers leave significant software performance on the table, even when compiling simple programs. In particular, Figure 1-1 shows that, although the C implementation (Version 3) is nearly 50 times faster than the Python implementation of the same code (Version 1), additional software-performance-engineering can increase the program's performance by another factor of 1000. Furthermore, compilers have thus far found limited success in exploiting parallelism effectively without programmer input [308, 417].

Rather than shelter programmers entirely from performance concerns, I believe that programming technologies can be developed to make software performance engineering accessible to average programmers. This thesis presents nine artifacts that work towards this goal by supporting principled approaches to reasoning about the behavior and efficiency of fast multicore software. Although these artifacts do develop simple programming models that encapsulate some performance concerns, these models do not solve the problem on their own. Theories of performance that are borne out in practice can enable programmers to use back-of-the-envelope calculations to predict the effect of a software change on efficiency before they write an optimized implementation. Efficient diagnostic tools can automate tasks in reasoning about program behavior and performance, augmenting a programmer's ability to reason about large and complex codebases. A simple framework, such as CSI, for creating new efficient diagnostic tools opens the door to developing tools that are tailored for a specific application or system. I believe that developing integrated programming technologies that support a science of fast code can enable average programmers to engage in software performance engineering in the post-Moore era.

The artifacts in this thesis develop and integrate programming technologies that support principled approaches to reasoning about software performance and efficiency, but more work remains to advance a coherent engineering science of fast code. We need additional simple programming models that support theories of performance that are borne out in practice. We need to embed these abstract models and theories into efficient diagnostic tools, compiler technology, and other parts of the software-development environment. We need to develop systems that provide visibility into the dynamic execution of a program and support careful measurement of that execution. And we need to educate programmers in how to use these software-performance-engineering technologies and how to think critically about software performance.

In summary, this thesis has shown that a more coherent science of fast code is feasible. In particular, we can build simple and integrated programming technologies that remedy the *ad hoc* and unprincipled nature of software performance engineering. Although developing a coherent science of fast code may be challenging, I believe that doing so will be crucial if we wish to see new transformative computing functionality emerge after Moore's Law ends.

Bibliography

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[2] L. Adams and J. M. Ortega. A multi-color SOR method for parallel computation. In *ICPP*, pages 53–56, 1982.

[3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[4] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP*, pages 183–193, 2007.

[5] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[6] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IPDPS*, pages 1–12, 2010.

[7] R. Agrawal and S. D. Stoller. Type inference for parameterized race-free Java. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. 2004.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Second edition, 2006.

[9] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[10] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38. 1999.

[11] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification Version 1.0*. Sun Microsystems, Inc., 2008.

[12] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A comparison of parallel graph coloring algorithms. Technical report, Syracuse University, 1995.

[13] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7:567–583, 1986.

[14] Amazon. Amazon Web Services (AWS) — cloud computing services [online]. 2016. URL: https://aws.amazon.com/ [cited July 20, 2016].

[15] Amazon Web Services. Announcing Amazon Elastic Compute Cloud (Amazon EC2) — beta [online]. 2006. URL: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/.

[16] Amazon Web Services. Amazon EC2 pricing [online]. 2016. URL: https://aws.amazon.com/ec2/pricing/ [cited July 20, 2016].

[17] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. In *SIGMETRICS*, pages 115–125, 1990.

[18] J. Ansel and C. Chan. PetaBricks: Building adaptable and more efficient programs for the multi-core era. *Crossroads, The ACM Magazine for Students (XRDS)*, 17(1):32–37, 2010.

[19] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *PACT*, 2014.

[20] Apache Software Foundation. APR's version numbering, 2015. URL: http://apr.apache.org/versioning.html.

[21] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, pages 197–206, 2008.

[22] E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 1987.

[23] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[24] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, pages 115–144, 2001.

[25] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[26] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[27] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *SIGKDD*, pages 44–54, 2006.

[28] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Keoster, E. Loh, K. Madduri, B. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2, 2007. Available at http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.2.doc.

[29] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and *st*-connectivity on the Cray MTA-2. In *ICPP*, pages 523–530, 2006.

[30] H. C. Baker, Jr. and Carl H. The incremental garbage collection of processes. *SIGPLAN Notices*, 12(8):55–59, 1977.

[31] D. R. Barach, D. H. Taenzer, and R. E. Wells. A technique for finding storage allocation errors in C-language programs. *SIGPLAN Notices*, 17(5):16–24, 1982.

[32] L. Barenboim and M. Elkin. Distributed (Δ+1)-coloring in linear (in Δ) time. In *STOC*, pages 111–120, 2009.

[33] R. Barik, Z. Budimlić, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero multicore software research project. In *OOPSLA*, pages 735–736, 2009.

[34] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT*, pages 41–52, 2009.

[35] R. Barik, J. Zhao, and V. Sarkar. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *PACT*, pages 29–40, 2013.

[36] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *ASE*, pages 106–115, 2003.

[37] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *SPAA*, pages 133–144, 2004.

[38] J. L. Bentley. *Writing Efficient Programs.* Prentice Hall, 1982.

[39] J. L. Bentley. *More programming pearls - confessions of a coder.* Addison-Wesley, 1988.

[40] S. Berchtold, C. Böhm, B. Braunmüller, D. A. Keim, and H.-P. Kriegel. Fast parallel similarity search in multimedia databases. In *SIGMOD*, pages 1–12, 1997.

[41] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[42] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, pages 81–96, 2009.

[43] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Inc., 1989.

[44] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[45] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81, 2008. `doi:10.1145/1454115.1454128`.

[46] C. Bienia and K. Li. Characteristics of workloads using the pipeline programming model. In *ISCA*, pages 161–171, 2010.

[47] W. Binder, A. Villazón, D. Ansaloni, and P. Moret. @J: Towards rapid development of dynamic analysis tools for the Java Virtual Machine. In *VMIL*, pages 4:1–4:9, 2009.

[48] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *CRYPTO*, pages 216–233, 1999.

[49] G. E. Blelloch. Prefix sums and their applications. Technical report, Carnegie Mellon University, 1990.

[50] G. E. Blelloch. NESL: A nested data-parallel language. Technical report, Carnegie Mellon University, 1992.

[51] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[52] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, pages 181–192, 2012.

[53] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *SPAA*, pages 308–317, 2012.

[54] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *SPAA*, pages 3–16, 1991.

[55] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. In *SPAA*, pages 249–259, 1997.

[56] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[57] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[58] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[59] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, University of Texas at Austin, 1998.

[60] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *HotPar*, pages 4–4, 2009.

[61] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78, 2008.

[62] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA*, pages 97–112, 2007.

[63] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pages 101–113, 2008.

[64] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA*, pages 56–69, 2001.

[65] D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[66] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[67] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, TX, USA, 1992.

[68] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[69] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *WADS*, pages 37–48, 1999.

[70] D. Brosius, T. Curdt, M. Dahm, and J. van Zyl. Byte code engineering library [online]. 2012. URL: `https://commons.apache.org/proper/commons-bcel/`.

[71] R. G. Brown. Dieharder: A random number test suite [online]. August 2011. URL: `http://www.phy.duke.edu/~rgb/General/dieharder.php`.

[72] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *FDDO-4*, 2001.

[73] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[74] H. Brunst, M. Winkler, W. E. Nagel, and H.-C. Hoppe. Performance optimization for large scale computing: The scalable VAMPIR approach. In *Computational Science — ICCS 2001*, volume 2074 of *Lecture Notes in Computer Science*, pages 751–760. 2001.

[75] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition, 2015.

[76] J. E. Burns. Mutual exclusion with linear waiting using binary shared variables. *ACM SIGACT News*, 10(2):42–47, 1978.

[77] M. Callaghan and D. Mituzas. Poor man's profiler [online]. 2009. URL: `http://poormansprofiler.org/` [cited August 18, 2016].

[78] D. Campbell. The coming in-memory database tipping point [online]. 2012. URL: `https://blogs.technet.microsoft.com/dataplatforminsider/2012/04/09/the-coming-in-memory-database-tipping-point/` [cited July 18, 2016].

[79] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[80] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *STOC*, pages 106–112, 1977.

[81] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10-11):576–594, 2012.

[82] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *PPPJ*, pages 51–61, 2011.

[83] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN*, pages 98–105, 1982.

[84] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[85] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

[86] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, 2000.

[87] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[88] P. Chatarasi, J. Shirako, and V. Sarkar. Polyhedral optimizations of explicitly parallel programs. In *PACT*, pages 213–226, 2015.

[89] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *SPAA*, pages 298–309, 1998.

[90] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.

[91] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, pages 676–687, 2011.

[92] P. D. Coddington. Random number generators for parallel computers. Technical report, Northeast Parallel Architectures Center, Syracuse University, Syracuse, New York, 1997.

[93] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *STOC*, pages 206–219, 1986.

[94] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.

[95] R. Colwell. The chip design game at the end of Moore's Law. In *HCS*, pages 1–16, 2013.

[96] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.

[97] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. Spidle: a DSL approach to specifying streaming applications. In *GPCE*, pages 1–17, 2003.

[98] S. Contini, R. L. Rivest, M. J. B. Robshaw, and Y. L. Yin. The security of the RC6 block cipher. 1998. URL: `http://people.csail.mit.edu/rivest/publications.html`.

[99] M. E. Conway. A multiprocessor system design. In *AFIPS*, pages 139–146, 1963.

[100] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[101] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[102] Rachel Courtland. The murky origins of "Moore's Law" [online]. April 2015. URL: `http://spectrum.ieee.org/tech-talk/semiconductors/devices/the-murky-origins-of-moores-law` [cited March 14, 2016].

[103] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[104] J. C. Culberson. Iterated greedy graph coloring and the difficulty landscape. Technical report, University of Alberta, 1992.

[105] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *PPoPP*, pages 1–12, 1993.

[106] J. S. Danaher, I-T. A. Lee, and C. E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, 2006.

[107] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

[108] B. De Bus, D. Chanet, B. De Sutter, L. Van Put, and K. De Bosschere. The design and implementation of FIT: A flexible instrumentation toolkit. In *PASTE*, pages 29–34, 2004.

[109] R. H. Dennard, F. H. Gaensslen, H.-N. Wu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[110] J. E. Dennis, Jr. and T. Steihaug. On the successive projections approach to least-squares problems. *SIAM Journal on Numerical Analysis*, 23(4):717–733, 1986.

[111] T. Denniston. D21752 Comprehensive Static Instrumentation (1/2): LLVM pass [online]. 2016. URL: `https://reviews.llvm.org/D21752` [cited August 11, 2016].

[112] T. Denniston. D21753 Comprehensive Static Instrumentation (2/2): Clang flag [online]. 2016. URL: https://reviews.llvm.org/D21753 [cited August 11, 2016].

[113] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ASPLOS*, pages 85–96, 2009.

[114] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A relaxed consistency deterministic computer. In *ASPLOS*, pages 67–78, 2011.

[115] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-on sound and complete race detection in software and hardware. In *ISCA*, pages 201–212, 2012.

[116] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *ICALP*, pages 235–246, 1992.

[117] K. Diks. A fast parallel algorithm for six-colouring of planar graphs. In *Mathematical Foundations of Computer Science*, volume 233 of *Lecture Notes in Computer Science*, pages 273–282. 1986.

[118] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.

[119] D. Dimitrov, M. Vechev, and V. Sarkar. Race detection in two dimensions. In *SPAA*, pages 101–110, 2015.

[120] C. Ding and H. Xu. Program locality analysis tools, 2015. URL: https://github.com/dcompiler/loca.

[121] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, pages 245–257, 2003.

[122] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP*, pages 1–10, 1990.

[123] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *PADD*, pages 85–96, 1991.

[124] A. C. Dinning. *Detecting Nondeterminism in Shared Memory Parallel Programs*. PhD thesis, Department of Computer Science, New York University, 1990.

[125] W. Du, R. Ferreira, and G. Agrawal. Compiler support for exploiting coarse-grained pipelined parallelism. In *SC*, pages 8–21, 2003.

[126] DWARF Standards Committee. DWARF debugging information format version 4. Available at http://dwarfstd.org/doc/DWARF4.pdf.

[127] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. In *STOC*, pages 174–183, 1993. URL: http://doi.acm.org/10.1145/167088.167145, doi:10.1145/167088.167145.

[128] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[129] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. OMPT: An OpenMP tools application programming interface for performance analysis. In *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 171–185. 2013.

[130] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *PADD*, pages 89–99, 1988.

[131] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[132] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[133] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *TCON*, pages 25–25, 1995.

[134] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[135] J. T. Fineman and C. E. Leiserson. Race detectors for Cilk and Cilk++ programs. In *Encyclopedia of Parallel Computing*, pages 1706–1719. 2011.

[136] M. Fischetti, S. Martello, and P. Toth. The fixed job schedule problem with spread-time constraints. *Operations Research*, 35(6):849–858, 1987.

[137] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[138] National Science Foundation. CISE research infrastructure: Mid-scale infrastructure – NSFCloud [online]. 2013. URL: `http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=504951`.

[139] P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith, and T. Warnock. Pseudo-random trees in Monte Carlo. *Parallel Computing*, 1(2):175–180, 1984.

[140] Free Software Foundation, Inc. GCC LinkTimeOptimization [online]. 2009. URL: `https://gcc.gnu.org/wiki/LinkTimeOptimization` [cited August 18, 2016].

[141] D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, 27(4):289–296, 1978.

[142] M. Frigo. A fast Fourier transform compiler. In *PLDI*, 1999.

[143] M. Frigo. A Cilk++ program for the knapsack challenge [online]. 2009. URL: `https://software.intel.com/en-us/courseware/249567` [cited August 18, 2016].

[144] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, pages 79–90, 2009.

[145] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, October 17–19 1999.

[146] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[147] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*, 1.14 edition, 2010. URL: `http://www.gnu.org/software/gsl/`.

[148] F. Garcia and J. Ferndandez. POSIX thread libraries. *Linux Journal*, 70, 2000. URL: `http://www.linuxjournal.com/article/3184`.

[149] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI*, pages 458–469, 2011.

[150] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[151] GCC team. GCC 4.9 release series changes, new features, and fixes [online]. 2014. URL: `https://gcc.gnu.org/gcc-4.9/changes.html`.

[152] GCC team. GOMP — an OpenMP implementation for GCC [online]. 2015. URL: `https://gcc.gnu.org/projects/gomp/`.

[153] A. H. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 12(12):1131–1146, 2000.

[154] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen. ColPack: Software for graph coloring and related problems in scientific computing. *ACM Transactions on Mathematical Software*, 40(1):1:1–1:31, 2013.

[155] A. E. Gelfand and A. F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85(410):398–409, 1990.

[156] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, 1984.

[157] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *PPoPP*, pages 43–52, 2008.

[158] P. B. Gibbons. A more practical PRAM model. In *SPAA*, pages 158–168, 1989.

[159] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[160] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[161] Gnome Project. *GLib 2.46 Reference Manual*, 2014. URL: `https://developer.gnome.org/glib/2.46/`.

[162] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988.

[163] M. Goldberg and T. Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing*, 18(2):419–427, 1989.

[164] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, 2(2):205–224, 1965. `arXiv:http://dx.doi.org/10.1137/0702016`.

[165] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[166] Google. Google Cloud Computing, Hosting Services & Cloud Support — Google Cloud Platform [online]. URL: `http://cloud.google.com/` [cited January 25, 2015].

[167] Google, Inc. Apache benchmark script, 2014. URL: `https://github.com/google/sanitizers/blob/master/thread-sanitizer/benchmarks/apache/run.sh`.

[168] Google, Inc. *Google C++ Style Guide*, 2015. URL: `https://google.github.io/styleguide/cppguide.html`.

[169] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, pages 151–162, 2006.

[170] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966.

[171] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. `gprof`: A call graph execution profiler. In *SIGPLAN*, pages 120–126, 1982.

[172] T. Grosser, A. Größlinger, and C. Lengauer. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[173] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *PPoPP*, pages 159–168, 1993.

[174] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[175] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *SPAA*, pages 166–177, 2014.

[176] W. C. Hasenplaugh. *Parallel Algorithms for Scheduling Data-Graph Computations*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2016.

[177] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter 1992 USENIX Conference*, pages 125–138, 1992.

[178] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. In *AFIPS*, pages 245–251, 1968.

[179] Y. He. Multicore-enabling discrete hedging in QuantLib [online]. 2009. URL: `http://software.intel.com/en-us/articles/multicore-enabling-discrete-hedging-in-quantlib/`.

[180] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156, 2010.

[181] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2006.

[182] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[183] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[184] W. W. Hines and D. C. Montgomery. *Probability and Statistics in Engineering and Management Science*. J. Wiley & Sons, third edition, 1990.

[185] F. L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Studies in Applied Mathematics*, 6(1-4):164–189, 1927.

[186] C. A. R. Hoare. Algorithm 63: Partition; Algorithm 64: Quicksort; and Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.

[187] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? Free will to choose. In *HPCA*, pages 333–334, 2011.

[188] 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.

[189] Intel Corporation. Intel® Cilk™ Plus samples [online]. URL: `https://software.intel.com/en-us/code-samples/intel-compiler/intel-compiler-features/intelcilkplus` [cited August 3, 2016].

[190] Intel Corporation. *Intel Cilk++ SDK Programmer's Guide*, October 2009. Document Number: 322581-001US.

[191] Intel Corporation. *Intel Cilk Plus Application Binary Interface Specification*, 2010. Document Number 324512-001US. URL: `https://software.intel.com/sites/products/cilk-plus/cilk_plus_abi.pdf`.

[192] Intel Corporation. Intrinsics for low overhead tool annotations. Document Number 326357-001US, 2011. URL: `https://www.cilkplus.org/open_specification/intrinsics-low-overhead-tool-annotations-v10`.

[193] Intel Corporation. Download Intel Cilk Plus software development kit [online]. 2012. URL: `https://software.intel.com/en-us/articles/download-intel-cilk-plus-software-development-kit/`.

[194] Intel Corporation. *Pin 2.11 User Guide*, 2012. URL: `https://software.intel.com/sites/landingpage/pintool/docs/49306/Pin/html/`.

[195] Intel Corporation. CilkPlus/LLVM [online]. 2013. URL: http://cilkplus.github.io/.

[196] Intel Corporation. *Intel® Cilk™ Plus Language Extension Specification, Version 1.2*, 2013. Document 324396-003US. URL: https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

[197] Intel Corporation. An introduction to the Cilk Screen race detector [online]. 2013. URL: https://software.intel.com/en-us/articles/an-introduction-to-the-cilk-screen-race-detector.

[198] Intel Corporation. Intel Cilk Plus [online]. 2015. URL: https://software.intel.com/en-us/intel-cilk-plus.

[199] Intel Corporation. Intel VTune Amplifier XE 2015 [online]. 2015. URL: http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[200] Intel Corporation. *Reference Manual for Intel Math Kernel Library 11.3-C*, 2015. URL: https://software.intel.com/en-us/mkl-reference-manual-for-c.

[201] Intel Corporation. Form 10-K (Annual Report). SEC filing, 2016. URL: http://files.shareholder.com/downloads/INTC/867590276x0xS50863-16-105/50863/filing.pdf.

[202] International Telecommunications Union. ICT Facts and Figures 2016 [online]. 2016. URL: http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2016.pdf [cited August 14, 2016].

[203] ITRS. International Technology Roadmap for Semiconductors 2.0, Executive Report [online]. 2015. URL: http://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2015/0_2015ITRS2.0ExecutiveReport.pdf.

[204] K. E. Iverson. *A Programming Language*. John Wiley & Sons, 1962.

[205] R. Jalan and A. Kejariwal. Trin-Trin: Who's calling? a Pin-based dynamic call graph extraction framework. *International Journal of Parallel Programming*, 40(4):410–442, 2012.

[206] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor. Kismet: Parallel speedup estimates for serial programs. In *OOPSLA*, pages 519–536, October 2011.

[207] D. Jeon, S. Garcia, C. Louie, S. K. Venkata, and M. B. Taylor. Kremlin: Like gprof, but for parallelization. In *PPoPP*, pages 293–294, 2011.

[208] P. G. Joisha, R. S. Schreiber, P. Banerjee, H. J. Boehm, and D. R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL*, pages 623–636, 2011.

[209] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[210] M. T. Jones and P. E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, pages 753–773, 1994.

[211] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. INSPIRE: The Insieme parallel intermediate representation. In *PACT*, pages 7–18, 2013.

[212] R. K. Gjertsen Jr., M. T. Jones, and P. E. Plassmann. Parallel heuristics for improved, balanced graph colorings. *Journal of Parallel and Distributed Computing*, 37(2):171–186, 1996.

[213] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. *ACM Transactions on Parallel Computing*, 3(1):2:1–2:31, 2016.

[214] R. M. Karp. A survey of parallel algorithms for shared-memory machines. Technical report, Berkeley, CA, USA, 1988.

[215] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[216] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. SPIRE, a sequential to parallel intermediate representation extension. Technical report, Technical Report CRI/A-487, MINES ParisTech, 2012.

[217] D. Khaldi, P. Jouvelot, F. Irigoin, C. Ancourt, and B. Chapman. Llvm parallel intermediate representation: Design and evaluation using openshmem communications. In *LLVM*, pages 2:1–2:8, 2015.

[218] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.

[219] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155, 2008.

[220] D. E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[221] D. E. Knuth. Structured programming with go to statements. In *Classics in Software Engineering*, pages 257–321. 1979.

[222] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1998.

[223] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, first edition, 1969.

[224] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[225] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.

[226] F. Kuhn. Weak graph colorings: Distributed algorithms and applications. In *SPAA*, pages 138–144, 2009.

[227] F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. In *PODC*, pages 7–15, 2006.

[228] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[229] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI*, pages 318–328, 1988.

[230] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, pages 291–300, 1995.

[231] C. Lasser and S. M. Omohundro. The essential *Lisp manual, release 1, revision 3. Technical Report 86.15, Thinking Machines, Cambridge, MA, 1986. URL: `http://omohundro.files.wordpress.com/2009/03/omohundro86_the_essential_starlisp_manual.pdf`.

[232] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.

[233] D. Lea. A Java fork/join framework. In *JAVA*, pages 36–43, 2000.

[234] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[235] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[236] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *USITS*, pages 73–82, 1997.

[237] I-T. A. Lee. *Memory Abstractions for Parallel Programming.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2012.

[238] I-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *PACT*, pages 411–420, 2010.

[239] I-T. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing*, 2(3):17:1–17:42, 2015. Special Issue for SPAA 2013.

[240] I-T. A. Lee and T. B. Schardl. Efficiently detecting races in Cilk programs that use reducer hyperobjects. In *SPAA*, pages 111–122, 2015. Invited to a special issue of *ACM Transactions on Parallel Computing.*

[241] I-T. A. Lee, A. Shafi, and C. E. Leiserson. Memory-mapping support for reducer hyperobjects. In *SPAA*, pages 287–297, 2012.

[242] J. Lee, S. P. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *LCPC*, pages 114–130, 1997. URL: `http://dl.acm.org/citation.cfm?id=645675.663619`.

[243] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Second Symposium on Large-Scale Digital Calculating Machinery*, pages 141–146, 1949.

[244] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes.* Morgan Kaufmann Publishers Inc., 1992.

[245] D. Leijen and J. Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. URL: `http://msdn.microsoft.com/magazine/`.

[246] C. E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, 2010.

[247] C. E. Leiserson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, T. B. Schardl, and N. C. Thompson. There's plenty of room at the top: What will drive growth in computer performance after moore's law ends? 2016. Unpublished manuscript.

[248] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, pages 303–314, 2010.

[249] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *PPoPP*, pages 193–204, 2012.

[250] J. Leskovec. SNAP: Stanford network analysis platform [online]. 2013. URL: `http://snap.stanford.edu/data/index.html`.

[251] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *PKDD*, pages 133–145, 2005.

[252] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[253] J. W. Lichtman, H. Pfister, and N. Shavit. The big data challenges of connectomics. *Nature Neuroscience*, 17(11):1448–1454, 11 2014.

[254] N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

[255] LLVM developer list. [LLVMdev] [cfe-dev] SPIR provisional specification is now available in the Khronos website [online]. 2012. URL: `http://lists.llvm.org/pipermail/llvm-dev/2012-September/053293.html` [cited November 20, 2015].

[256] LLVM developer list. [LLVMdev] [RFC] OpenMP Representation in LLVM IR [online]. 2012. URL: `http://lists.llvm.org/pipermail/llvm-dev/2012-September/053861.html` [cited November 20, 2015].

[257] LLVM developer list. [LLVMdev] LLVM Parallel IR [online]. 2015. URL: `http://lists.llvm.org/pipermail/llvm-dev/2015-March/083314.html` [cited November 20, 2015].

[258] LLVM developer list. [llvm-dev] RFC: Comprehensive Static Instrumentation [online]. 2016. URL: http://lists.llvm.org/pipermail/llvm-dev/2016-June/101162.html [cited August 11, 2016].

[259] LLVM Project. *LLVM Language Reference Manual*, 2015. URL: http://llvm.org/docs/LangRef.html.

[260] LLVM Project. LLVM link time optimization: Design and implementation, 2015. URL: http://llvm.org/docs/LinkTimeOptimization.html.

[261] LLVM Project. *LLVM's Analysis and Transform Passes*, 2015. URL: http://llvm.org/docs/Passes.html.

[262] LLVM Project. OpenMP®: Support for the OpenMP language [online]. 2015. URL: http://openmp.llvm.org/.

[263] L. Lovász, M. Saks, and W. T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Mathematics*, 75(1–3):319–325, 1989.

[264] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[265] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, 2010.

[266] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[267] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[268] S. MacDonald, D. Szafron, and J. Schaeffer. Rethinking the pipeline as object-oriented states with transformations. In *HIPS*, pages 12–21, 2004.

[269] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[270] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A domain-specific language for bytecode instrumentation. In *AOSD*, pages 239–250, 2012.

[271] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH*, pages 896–907, 2003.

[272] D. Marx. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica, Electrical Engineering*, 48(1):11–16, 2004.

[273] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26(3):436–461, 2000.

[274] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[275] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, 1983. URL: `http://doi.acm.org/10.1145/2402.322385`.

[276] A. McCallum. Cora data set [online]. 2012. URL: `http://people.cs.umass.edu/mccallum/data.html`.

[277] M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[278] D. McCrady. Avoiding contention using combinable objects [online]. 2008. URL: `https://blogs.msdn.microsoft.com/nativeconcurrency/2008/09/25/avoiding-contention-using-combinable-objects/` [cited August 19, 2016].

[279] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX ATC*, pages 279–294, 1996.

[280] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *SC*, pages 24–33, 1991.

[281] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *PADD*, pages 129–139, 1993.

[282] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *ICPP*, pages 105–113, 1990.

[283] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *PGAS*, 2011.

[284] T. Mitchell. NPIC500 data set [online]. 2009. URL: `http://www.cs.cmu.edu/tom/10709_fall2009/NPIC500.pdf`.

[285] J. Mitchem. On various algorithms for estimating the chromatic number of a graph. *The Computer Journal*, 19(2):182–183, 1976.

[286] E. F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching*, pages 285–292, 1959.

[287] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[288] G. E. Moore. Progress in digital integrated electronics. In *International Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[289] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, 1995.

[290] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[291] S. K. Muller and U. A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *SPAA*, pages 71–82, 2016.

[292] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *UAI*, pages 467–475, 1999.

[293] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.

[294] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*, 2008. Federal Information Standards Publication 180-3. URL: `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.

[295] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical modeling of pipeline parallelism. In *PACT*, pages 281–290, 2009.

[296] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[297] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[298] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[299] D. Nguyen, A. Lenharth, and K. Pingali. Deterministic Galois: On-demand, portable and parameterless. In *ASPLOS*, pages 499–512, 2014.

[300] K. Nigam and R. Ghani. Analyzing the effectiveness and applicability of co-training. In *CIKM*, pages 86–93, 2000.

[301] D. Novillo, R. Unrau, and J. Schaeffer. Concurrent ssa form in the presence of mutual exclusion. In *ICPP*, pages 356–364, 1998.

[302] I. Nudler and L. Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, 1986.

[303] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, pages 167–178, 2003.

[304] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, pages 97–108, 2009.

[305] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, 2008. URL: `http://www.openmp.org/mp-documents/spec30.pdf`.

[306] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.0*, 2013. URL: `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[307] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38*, pages 105–118, 2005.

[308] D. Padua. Parallelization, automatic. In *Encyclopedia of Parallel Computing*, pages 1442–1450. 2011.

[309] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA*, pages 348–354, 1984.

[310] S. S. Patil. Closure properties of interconnections of determinate systems. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 107–116. 1970.

[311] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. In *STOC*, pages 1–10, 2011.

[312] D. Patterson. An interview with stanford university president john hennessy. *Communications of the ACM*, 59(3):40–45, 2016.

[313] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc., 1988.

[314] D. E. Pettenò and D. J. Cozatt. Autotools mythbuster [online]. 2013. URL: `https://autotools.io/` [cited August 19, 2016].

[315] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of parallelism in algorithms. In *PLDI*, pages 12–25, 2011.

[316] A. Pop and A. Cohen. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *CPC*, 2010.

[317] A. Pop and A. Cohen. A stream-computing extension to OpenMP. In *HiPEAC*, pages 5–14, 2011.

[318] E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded c++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[319] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems*, 33(1):3:1–3:55, 2011.

[320] C. Prokopp. GraphChi: How a Mac Mini outperformed a 1,636 node Hadoop cluster [online]. 2014. URL: `http://www.semantikoz.com/blog/graphchi-mac-mini-outperformed-1636-node-hadoop-cluster/` [cited July 22, 2016].

[321] W. Pugh. Fixing the Java memory model. In *JAVA*, pages 89–98, 1999.

[322] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[323] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, Seattle, WA, 2013.

[324] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 368–383. 2010.

[325] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *PLDI*, pages 531–542, 2012.

[326] A. G. Ranade. The delay sequence argument. In *Handbook of Randomized Algorithms*, chapter 1. Kluwer Academic Publishers, 2001.

[327] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT*, pages 177–188, 2004.

[328] E. C. Reed, N. Chen, and R. E. Johnson. Expressing pipeline parallelism using TBB constructs: a case study on what works and what doesn't. In *SPLASH*, pages 133–138, 2011.

[329] J. Reinders. *VTune Performance Analyzer Essentials.* Intel Press, 2005.

[330] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly Media, Inc., 2007.

[331] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, 1998.

[332] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 block cipher [online]. 1998. URL: `http://people.csail.mit.edu/rivest/publications.html`.

[333] R. Rojas. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, 19(2):5–16, 1997.

[334] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *NT*, pages 1–7, 1997.

[335] E. Ruf. Effective synchronization removal for Java. In *PLDI*, pages 208–218, 2000.

[336] R. Rugina and M. C. Rinard. Pointer analysis for structured parallel programs. *ACM Transactions on Programming Languages and Systems*, 25(1):70–116, 2003.

[337] Y. Saad. *SPARSKIT: A basic toolkit for sparse matrix computations.* Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990.

[338] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *SC*, pages 16:1–16:12, 2011.

[339] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *PACT*, pages 22–32, 2011.

[340] A. E. Sariyüce, E. Saule, and Ü. V. Çataryürek. Improving graph coloring on distributed-memory parallel computers. In *HiPC*, pages 1–10, 2011.

[341] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *LCPC*, pages 94–113, 1997.

[342] V. Sarkar and B. Simons. Parallel program graphs and their classification. In *LCPC*, pages 633–655, 1994.

[343] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *SOSP*, pages 27–37, 1997.

[344] T. B. Schardl. Design and analysis of a nondeterministic parallel breadth-first search algorithm. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2010.

[345] T. B. Schardl, T. Denniston, D. Doucet, B. C. Kuszmaul, I-T. A. Lee, and C. E. Leiserson. Comprehensive static instrumentation for dynamic-analysis tools. Unpublished manuscript, 2016.

[346] T. B. Schardl, B. C. Kuszmaul, I-T. A. Lee, W. M. Leiserson, and C. E. Leiserson. The Cilkprof scalability profiler. In *SPAA*, pages 89–100, 2015.

[347] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM IR. Submitted for publication, 2016.

[348] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford University Computing Lab., 1971.

[349] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1996.

[350] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, pages 309–318, 2012.

[351] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer — data race detection in practice. In *WABI*, pages 62–71, 2009.

[352] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with LLVM compiler. Technical Report 37278, Google, 2011.

[353] N. Shavit. A multicore path to connectomics-on-demand. In *SPAA*, pages 211–211, 2016.

[354] S. Shende, A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan. Portable profiling and tracing for parallel, scientific applications using C++. In *SPDT*, pages 134–145, 1998.

[355] S. S. Shende and A. D. Malony. The Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[356] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: a new reduction construct for dynamic parallelism. In *IPDPS*, pages 1–12, 2009.

[357] J. Shun. *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2015.

[358] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[359] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *SPAA*, pages 152–163, 2013.

[360] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, pages 68–70, 2012.

[361] J. Shun, Laxman D., and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*, pages 403–412, 2015.

[362] P. Singla and P. Domingos. Entity resolution with Markov logic. In *ICDM*, pages 572–582, 2006.

[363] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, 1991.

[364] H. Srinivasan and D. Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical Report CU-CS-564-91, University of Colorado at Boulder, 1991.

[365] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment for explicitly parallel programs. In *POPL*, pages 260–272, 1993.

[366] H. Srinivasan and M. Wolfe. Analyzing programs with explicit parallelism. In *LCPC*, pages 405–419, 1991.

[367] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *PLDI*, pages 196–205, 1994.

[368] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. Technical Report 92/6, Digital Western Research Laboratory, 1992.

[369] R. M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection (for GCC version 6.1.0)*. Free Software Foundation, Inc., 2016.

[370] G. L. Steele Jr. Making asynchronous parallelism safe for the world. In *POPL*, pages 218–231, 1990.

[371] G. L. Steele Jr., D. Lea, and C. H. Flood. Fast splittable pseudorandom number generators. In *OOPSLA*, pages 453–472, 2014.

[372] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX*, pages 1–9, 1992.

[373] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler. Flexible software profiling of GPU architectures. In *ISCA*, pages 185–197, 2015.

[374] J. Stoer, R. Bulirsch, R. H. Bartels, W. Gautschi, and C. Witzgall. *Introduction to Numerical Analysis*. Springer, New York, 2002.

[375] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[376] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, fourth edition, 2013.

[377] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *SIGMETRICS*, pages 24–35, 1993.

[378] J. Sukha. Cilkpub: A library of community-contributed Cilk Plus code [online]. 2013. URL: `https://software.intel.com/en-us/forums/intel-cilk-plus/topic/384639`.

[379] J. Sukha. Piper: Experimental support for parallel pipelines in Intel® Cilk™ Plus [online]. 2013. URL: `https://www.cilkplus.org/sites/default/files/experimental-software/PiperReferenceGuideV1.0_0.pdf`.

[380] J. Sukha. Brief announcement: A compiler-runtime application binary interface for pipe-while loops. In *SPAA*, pages 83–85, 2015.

[381] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *PACT*, pages 147–156, 2010.

[382] M. Szegedy and S. Vishwanathan. Locality based graph coloring. In *STOC*, pages 201–207, 1993.

[383] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPoPP*, pages 229–240, 2009.

[384] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[385] GCC Team. GCC 4.8 release series changes, new features, and fixes [online]. 2014. URL: `https://gcc.gnu.org/gcc-4.8/changes.html`.

[386] Telecommunication Standardization Sector. High efficiency video coding. Standard H.265, International Telecommunication Union, 2014.

[387] R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, 1976.

[388] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO*, pages 356–369, 2007.

[389] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *SODA*, pages 615–624, 2004.

[390] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ISSTA*, pages 86–96, 2002.

[391] A. M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[392] R. Utterback, K. Agrawal, J. T. Fineman, and I-T. A. Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *SPAA*, pages 83–94, 2016.

[393] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[394] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, pages 18–34, 2000.

[395] A. van Heukelum, G. T. Barkema, and R. H. Bisseling. Dna electrophoresis studied with the cage model. *Journal of Computational Physics*, 180(1):313–326, 2002.

[396] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York, Inc., 1999.

[397] W. von Hagen. *The Definitive Guide to GCC*, chapter 6. Apress, second edition, 2006.

[398] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.

[399] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *ESEC-FSE*, pages 205–214, 2007.

[400] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, pages 521–530, 2005.

[401] D. W. Wall. Link-time code modification. Technical Report 89/17, Digital Western Research Laboratory, 1989.

[402] M. N. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.

[403] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[404] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *SC*, pages 1–27, 1998.

[405] D. A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size [online]. 2001. URL: `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`.

[406] A. White. Serp [online]. 2011. URL: `http://serp.sourceforge.net/`.

[407] J. Widendorfer. Sequential performance analysis with Callgrind and KCachegrind. In *Tools for High Performance Computing*, pages 93–113, 2008.

[408] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[409] M. Wimmer. Wait-free hyperobjects for task-parallel programming systems. In *IPDPS*, pages 803–812, 2013.

[410] W. A. Wulf. A case against the GOTO. In *Classics in Software Engineering*, pages 83–98. 1979.

[411] W. A. Wulf, R. K. Johnson, C. B. Weinstock, and C. M. Geschke. *The design of an optimizing compiler*. American Elsevier Pub. Co., 1975.

[412] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *ASPLOS*, pages 343–356, 2013.

[413] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and Ü. Çatalyürek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *SC*, page 25, 2005.

[414] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, pages 325–336, 2009.

[415] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.

[416] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *SC*, pages 712–721, 1991.

[417] G. Zhang, P. Unnikrishnan, and J. Ren. Experiments with auto-parallelizing SPEC2000FP benchmarks. In *LCPC*, pages 348–362. 2005.

[418] Y. Zhang and E. A. Hansen. Parallel breadth-first heuristic search on a shared-memory architecture. In *AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, 2006.

[419] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, pages 427–440, 2012.

[420] J. Zhao and V. Sarkar. Intermediate language extensions for parallelism. In *SPLASH*, pages 329–340, 2011.