# Executing Multithreaded Programs Efficiently

by

Robert D. Blumofe

Sc.B., Brown University (1988)
S.M., Massachusetts Institute of Technology (1992)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1995



Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
August 18, 1995

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

# Executing Multithreaded Programs Efficiently

by

Robert D. Blumofe

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

This thesis presents the theory, design, and implementation of *Cilk* (pronounced "silk") and *Cilk-NOW*. Cilk is a C-based language and portable runtime system for programming and executing multithreaded parallel programs. Cilk-NOW is an implementation of the Cilk runtime system that transparently manages resources for parallel programs running on a network of workstations.

Cilk is built around a provably efficient algorithm for scheduling the execution of *fully strict* multithreaded computations. Based on the technique of *work stealing*, this algorithm achieves time, space, and communication bounds that are all within a small constant factor of optimal. Using these performance guarantees, Cilk provides a simple abstraction of performance that allows programmers to predict accurately how program changes will affect execution time.

Cilk-NOW provides high-level resource management services so that Cilk programs can run efficiently on a network of workstations. These services include *adaptive parallelism* and *fault tolerance*. With adaptive parallelism, programs execute on a pool of workstations that dynamically grows and shrinks in response to both the availability of "idle" machines and the availability of parallelism within the program. Cilk-NOW also provides transparent fault tolerance with a fully distributed checkpointing mechanism.

**Keywords:** parallel computing, distributed computing, dataflow, multithreaded programs, parallel languages, parallel runtime systems, Cilk, strict execution, stack memory, thread scheduling, scheduling algorithms, randomized algorithms, delay sequence, work stealing, computer chess, massively parallel supercomputer, network of workstations, adaptive parallelism, fault tolerance.

**Thesis Supervisor:** Charles E. Leiserson
**Title:** Professor

# Acknowledgments

I remember the first time I met Charles Leiserson and discussed possible research projects. He spoke of some grand ideas, and he waved his arms a lot. Something about trees. We were going to solve some important problems together. Only one difficulty: I had no idea what he was talking about. For better or worse, I now seem to understand him, and our collaboration did produce some results that I am very proud of. Charles has a terrific sense of research direction, and rather than push me in any direction, he pulled. Rather than tell me where to go, he showed the way. All the while, Charles left me room to be creative and pursue paths that I found myself. He supported me in all my research pursuits with his own enthusiasm and with a generous allocation of resources, both human and machine. Charles also has a great sense of humor, and we had fun, even giggling like children one afternoon as we imagined how much more fun we would have playing the recycling game (see Section 4.2) instead of hockey. Charles tells me that now I speak in grand ideas and wave my arms a lot. Could be worse.

Besides serving as my advisor, collaborator, and friend, Charles also showed me how to clarify my writing with focused work and patience. It did not come easy. After one particularly long and grueling session of writing (Charles has infinite patience), I left Charles' office and the first person I stumbled upon was Tom Cormen. With a sigh, I asked him, "have you ever written a paper with Charles?" He just looked at me like that is the stupidest question I've ever asked him (see [33]). It isn't.

Much of the work in this thesis could not have been done without the collaboration of the entire Cilk team. Led by Charles, this crackerjack team includes or has included Shail Aditya now of Hewlett Packard, Matteo Frigo, Michael Halbherr now of the Boston

The research reported in this thesis was supported in part by the Advanced Research Projects Agency (ARPA) under Grants N00014-94-1-0985 and N00014-92-J-1310. In addition, I was generously supported throughout my graduate career by a National Science Foundation (NSF) Graduate Student Fellowship and an ARPA High-Performance Computing Graduate Fellowship administered through the University of Maryland Institute for Advanced Computer Study.

Consulting Group, Chris Joerg, Bradley Kuszmaul now of Yale University, Phil Lisiecki, Howard Lu, Rob Miller now of Carnegie Mellon University, David Park, Keith Randall, Larry Rudolph, Sivan Toledo now of IBM, and Yuli Zhou now of AT&T Bell Laboratories.

Many people at MIT have made significant contributions to the work in my thesis. Much of the combinatorial technique and the delay-sequence argument used in Chapter 4, I learned from Tom Leighton. Bonnie Berger helped with some of my early probabilistic analysis. Esther Jesurum and Ethan Wolf helped with the lower bound. Eric Brewer now of the University of California at Berkeley was my collaborator in some of my early forays into system building. Others who have lent inspiration and ideas include Arvind, Tom Cormen now of Dartmouth College, Frans Kaashoek, Arthur Lent, Greg Papadopoulos now of Sun, and Bill Weihl. Scott Blomquist kept the CM5 and our network of workstations up and running. David Jones provided extensive help with some of the subtleties of LaTeX. Others who have helped make my life easy (and fun) include Be Hubbard, Irena Kveraga, and Cheryl Patton.

Several people from outside MIT also made significant contributions to the work in this thesis. Bruce Maggs of Carnegie Mellon University outlined the delay sequence argument, and Greg Plaxton of the University of Texas at Austin gave technical comments on the recycling game analysis. Others who contributed ideas include Guy Blelloch of Carnegie Mellon University, Warren Burton of Simon Fraser University, and Yanjun Zhang of Southern Methodist University.

I would not have made it to MIT from Brown (where I was an undergraduate) had it not been for some of the inspiring professors in Brown's math and computer science departments. Tom Banchoff first turned me on to computer graphics; Jeff Vitter, now of Duke University, first turned me on to theory and algorithms; John Savage first turned me on to parallel computation; and Andy van Dam first turned me on to my future in computation. I took my first computer science class from Andy, and I took my first adventure into research in Andy's graphics group. From this beginning, Andy infected me with his exuberant sense of excitement and wonder over a future in which anything can happen when computation meets imagination. I was a less than model student in my early years at Brown, and I know I let him down on more than one occasion, but Andy must have seen something that nobody else could, because he continued to take risks with me, presenting me with one opportunity after another. When I took a year off from Brown, Andy got me a great job at a small startup company called Cadre Technologies, and later it was Andy who first suggested that I apply

to graduate school and then backed up the suggestion with a strong letter of recommendation. On those last two opportunities, I think I finally came through. I don't know why Andy took such a personal interest in me or why he was so unwaivering in his confidence, but I think about him often and with gratitude and affection.

My years to date have been filled with humor, adventure, satisfaction, and fun, because I have had the good fortune of good friends, some of whom I shall now mention ordered from oldest to most recent. Danny Klein: Danny was my summer brother and best friend for many years. The gang out at Horton Lakes, including the Horton family—Jack, Gretchen, and John—Greg Cooper, and many others who have come and gone: It has been my privilege and joy to waterski with these folks at their world-class site. Max Kennedy: I've never met anyone with more energy, less fear, or a bigger heart, and hey man, Max and I had some serious fun. Jeff Cutler: Jeff and I drove cross country four times, discovered the precipice, watched for suicide deer, went to Pete Freehafer's wedding as the Goldberg brothers, and had countless other silly adventures. I can't even write this without laughing. Liz Cohen: Liz and I seem to have an unbreakable bond forged through some strange days and stranger nights. Anna Hedly: Everyone who knows me well knows how important Anna was and is to me. We rarely talk or see each other, but I still think about her often. Also, Anna got me started as a pilot. Janet Barber: Janet is probably my best friend and the person I most enjoy talking to and hanging around with. Going to Dead shows and living large with Janet in Vegas was surely the best possible way to turn thirty. The gang from Motion Dynamics—Slim, Dave, Mike, and Leo: These guys fixed my little race car whenever I wadded it up in a ball (often), and they were a hell of a lot of fun to hang with. Former and present-day Cadroids—Lou Bershad, Reed Flemming, Alan Hecht, Lou Mazzuchelli, Lou Reynolds, and others: They were and still are a talented and fun group of people to work and play with. The entire Bar-Yam family—Miriam, Zvi, Yaneer, Aureet, and Sageet: The Bar-Yams first were my landlords in Cambridge but foremost are my cherished friends. Esther Jesurum: Esther got me started playing hockey and was a fun waterskiing partner. Helen Greiner: Helen was always up for some fun. She would do anything except wear a silly hat.

My family is somewhat distant and rather peculiar but also quite wonderful, and I thank them for their support and tolerance. My older brother Michael was my role model growing up, and he inspired my interest in math and science. Michael is now an emergency room doctor and saves lives every day. (On a good day, I can improve an algorithm.) My older sister Maria first showed me the adventurous, and occasionally dangerous, side of life. My younger sister Joanna is my closest relative spatially (she lives nearby in the Boston area), temporally (we are one year and six days apart in age), and emotionally. Joanna is married to Brad Meiseles and has two sons Benjamin and Brandon. I have really enjoyed having family, especially Benjamin and Brandon, nearby. My mother gave me tremendous confidence and high expectations, and these qualities have brought me through some difficult times. I think I get the rational side of my character from her. My father came to parenthood rather late in life, but I don't know that anyone ever worked at it harder or enjoyed it more. I think I get the emotional side of my character from him. I hope I can be as well liked and well respected in my profession as he seems to have been in his.

ROBERT D. BLUMOFE<br>
*Cambridge, Massachusetts*<br>
*August 18, 1995*

# Contents

# List of Figures

# Chapter 1

# Executing multithreaded programs efficiently

In writing a parallel application in a multithreaded language, a programmer expresses parallelism by coding instructions in a partial execution order, and the programmer or the compiler partitions these instructions into totally ordered sequences called *threads*. The programmer need not specify which processors of a parallel computer execute which threads nor exactly when each thread should be executed. These scheduling decisions are made automatically by the runtime system's scheduler, and the program only requires that each instruction of each thread is executed by some processor at a time consistent with the partial order. Nevertheless, if the programmer is to relinquish control over these scheduling decisions to the runtime system, then the runtime system must guarantee that it will make good scheduling decisions to execute the program efficiently and with predictable performance. In this thesis, we develop algorithmic foundations for the efficient scheduling of multithreaded programs, and we build a multithreaded language and runtime system on top of this foundation. In both algorithmic and empirical work, we show that for a large and important class of multithreaded programs, a runtime system can deliver efficient and predictable performance, guaranteed.

## 1.1 Scheduling multithreaded programs

A multithreaded language provides programmers with a means to create and synchronize multiple computational threads, and the runtime system for such a language automatically schedules the execution of these threads on the processors of a parallel computer. To execute a multithreaded program efficiently, the runtime system's scheduler must keep the processors busy doing work in order to realize parallel speedup, and simultaneously, it must maintain memory usage to within reasonable limits and avoid interprocessor communication to the greatest extent possible. In this section, we give an overview of the *Cilk* (pronounced "silk") multithreaded language and runtime system. The Cilk runtime system's scheduler is built on an algorithmic foundation and is provably efficient with respect to time, space, and communication. We shall also overview *Cilk-NOW*, an implementation of the Cilk runtime system for networks of workstations. Cilk-NOW leverages properties of Cilk's scheduling algorithm in order to run Cilk programs efficiently in the dynamic and potentially faulty environment of a network of workstations.

Consider a program that uses double recursion to compute the Fibonacci function. The Fibonacci function $\mathrm{fib}(n)$ for $n \geq 0$ is defined as

$$\mathrm{fib}(n) = \begin{cases} n & \text{if } n < 2; \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{otherwise.} \end{cases}$$

Figure 1.1 shows how this function is written in C and in Cilk. While double recursion is a terrible way to compute Fibonacci numbers, this toy example does illustrate a common pattern occurring in divide-and-conquer applications: recursive calls solve smaller subcases and then the partial results are merged to produce the final result. Notice that other than the

```
int Fib (int n)
{  if (n<2)
      return n;
   else
   {  int x, y;
      x = Fib (n-1);
      y = Fib (n-2);

      return (x+y);
   }
}
```

**(a)** C function.

```
cilk int Fib (int n)
{  if (n<2)
      return n;
   else
   {  int x, y;
      x = spawn Fib (n-1);
      y = spawn Fib (n-2);
      sync;
      return (x+y);
   }
}
```

**(b)** Cilk procedure.

**Figure 1.1**: A C function and a Cilk procedure to compute the $n$th Fibonacci number.

`cilk` keyword which identifies a Cilk procedure, the Cilk version differs from its C counterpart only in its use of `spawn` and `sync`. The `spawn` keywords turn the recursive calls into recursive *spawns*. A spawn is the parallel analogue of a procedure call. The instructions executed by `Fib` form a thread, and a spawn creates a new child thread that may execute concurrently with its parent. Thus, when `Fib` performs the two recursive spawns, the spawned child threads may execute concurrently. After performing the spawns, `Fib` does a `sync` to synchronize with its children. When `Fib` gets to the `sync` statement, it must suspend and wait until its two spawned children return. Only then may `Fib` proceed to add the return values. Thus, the spawns and synchronizations specify a partial execution order on the program's instructions.

The spawned threads must be executed by the processors of a parallel machine in a manner consistent with the program-specified partial order, and in general, the scheduling of threads to processors must be done at runtime. For many programs, the spawning of threads depends on the input data, and for these programs the threads cannot even be identified until runtime. Furthermore, even if the threads can be statically identified, determining the execution time of any given thread may be as difficult as actually running the thread, so load-balancing considerations may dictate runtime scheduling. For these types of applications, we must separate the static expression of parallelism in the program from the dynamic scheduling of threads at runtime. A multithreaded language permits this separation by incorporating a thread scheduler in its runtime system.

The Cilk language supports this separation with a set of extensions to the C language for expressing parallelism and a runtime system that automatically exploits the program's parallelism on a parallel computer. Cilk applications coded to date include protein folding, graphic rendering, backtrack search, and the `⋆Socrates` chess program, which won second prize in the 1995 ICCA World Computer Chess Championship. The Cilk runtime system is particularly easy to implement and easily portable. In fact, in several cases, individuals unfamiliar with Cilk have ported the runtime system to new machines in under 1 week. Currently, the Cilk runtime system runs on the Connection Machine CM5 MPP, the Intel Paragon MPP, the Sun SparcStation SMP, the Silicon Graphics Power Challenge SMP, and the Cilk-NOW network of workstations, discussed below.

The Cilk runtime system automatically manages the low-level details of thread scheduling, and it does so with a "work-stealing" scheduler that is provably efficient. Writing a high-performance parallel application in Cilk, the programmer can focus on expressing the parallelism in the algorithm, insulated from scheduling details and secure in the knowledge that the runtime system delivers guaranteed efficient performance. Figure 1.2 plots measured speedup values for a large number of runs of the `⋆Socrates` chess program on the CM5. Both axes are normalized as we shall explain when we revisit this speedup plot in Section 5.2. For now, we think of the normalized machine size as the number $P$ of processors increasing from left to right, and we think of the normalized speedup simply as speedup—the ratio of the 1-processor execution time to the $P$-processor execution time. The 45-degree line and the horizontal line at 1.0 are upper bounds on the achievable speedup. This plot

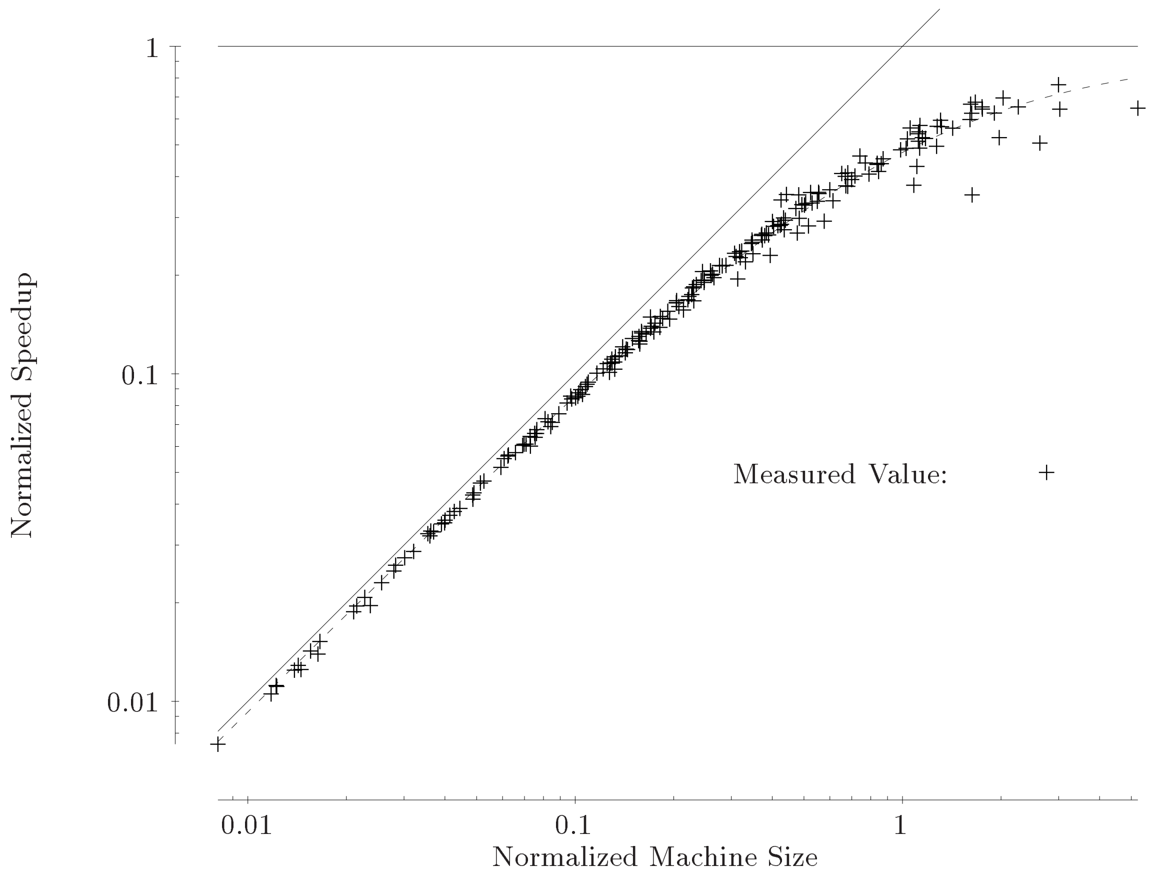



**Figure 1.2**: Measured speedups for the ⋆`Socrates` chess program.

shows that every run achieved near optimal speedup. This performance is not the good fortune of a single application. Rather, we prove that for a large and important class of programs, the Cilk runtime system achieves near optimal performance, guaranteed. Moreover, Cilk gives the user an algorithmic model of application performance based on the measures of *work*—the total number of instructions executed—and *critical path length*—the length of a longest path in the partial order of instructions—which can be used to predict the runtime of a Cilk program accurately. Notice that the measured speedup values in Figure 1.2 all cluster on a curve. As we shall see in Section 5.2, this curve is defined as a function of only three parameters: work, critical path length, and the number of processors. Consequently, a Cilk programmer can tune the performance of his or her application by focusing on the work and critical path, unworried about machine specific performance details.

In order to execute Cilk programs efficiently on a network of workstations, the Cilk-NOW runtime system implements "adaptive parallelism" and transparent fault tolerance. *Adaptive parallelism* allows a Cilk program to take advantage of otherwise-idle machines whether or not they are idle when the program starts running and whether or not they will remain otherwise idle for the duration of the run. When a given workstation is left idle by its owner, it automatically joins in and helps out with the execution of a Cilk program. When the owner returns to work, the machine automatically retreats from the Cilk program. The criteria to determine when a machine is idle can be customized for each machine. We believe that maintaining the owner's sovereignty is essential if we want owners to allow their machines to be used for parallel computation. With transparent fault tolerance built into the Cilk-NOW runtime system, Cilk jobs may survive machine crashes or network outages despite the fact that Cilk programs are *fault oblivious*, having been coded with no special provision for handling machine or network failures. If a worker crashes, then other workers automatically redo any work that was lost in the crash. In the case of a more catastrophic failure, such as a power outage or a total network failure in which all workers may crash, Cilk-NOW provides automatic checkpointing, so when service is restored, the Cilk job may be restarted with minimal lost work.

Recently, we ran a Cilk protein-folding application using Cilk-NOW on a network of about 50 Sun SparcStations connected by Ethernet to solve a large-scale problem. The program ran for 9 days, survived several machine crashes and reboots, utilized 6566 processor-hours of otherwise-idle cycles, with no administrative effort on our part, while other users of the network went about their business unaware of the program's presence. Adaptive parallelism and fault tolerance were invaluable. In contrast, running the same experiment on the CM5 required manually breaking the job into pieces small enough to complete in the interval of time between repartitionings and failures of the machine and then manually combining results. These pieces were fed to the machine through shell scripts to ensure that the machine executed these pieces even when we were not around to perform manual feeding.

The efficient and predictable performance of the Cilk runtime system and the adaptive parallelism and fault tolerant features of Cilk-NOW are possible because Cilk is built on a theoretically sound foundation. In particular, the Cilk programming model is highly structured and the Cilk runtime system's scheduler employs a provably efficient scheduling algorithm.

In establishing an algorithmic foundation for scheduling multithreaded computations, we have proven a lower bound showing that in general there exist multithreaded computations for which efficient scheduling is impossible, and we have proven an upper bound showing that for an important class of computations, efficient scheduling is possible. We show that in general, there exist multithreaded computations with vast amounts of provably useless parallelism. For such a computation, any execution schedule attempting to exploit this parallelism must use an amount of space per processor vastly in excess of that required by a 1-processor execution. On the other hand, we show that for the class of *strict* computations, all of the parallelism can be exploited. Intuitively, a strict computation is one in which threads only return values and synchronize with their ancestor threads. Computations, such as those derived from divide-and-conquer, backtracking search, branch-and-bound, game-tree search, and many other programs, are naturally strict and have large amounts of parallelism. We show that for any strict computation and any number of processors, there exists an execution schedule with execution time that is within a factor of 2 of optimal while using no more space per processor than that required by a 1-processor execution. We give a simple, though impractical and nonscalable, algorithm to compute such efficient schedules.

For practical and scalable application, we give a simple and fully distributed online algorithm for scheduling the execution of multithreaded computations. This algorithm is based on the technique of random *work stealing* in which processors needing work steal threads from other processors chosen at random. We show that for "fully strict" (well-structured) computations, this algorithm is simultaneously efficient with respect to time, space, and communication. In particular, this work-stealing algorithm achieves execution time that is universally optimal to within a constant factor, execution space that is existentially optimal to within a constant factor, and communication that is also existentially optimal to within a constant factor. This communication bound confirms the folk wisdom that work-stealing algorithms require much less communication than their "work-sharing" counterparts. In our analysis, we conservatively assume that when multiple processors simultaneously attempt to access a single data structure, then their accesses are serialized in an order determined by an adversary. These bounds are the first proven bounds for the case of computations with synchronization. The Cilk runtime system's scheduler implements this work-stealing algorithm.

## 1.2 Previous results and related work

The Cilk runtime system differs from other systems for parallel multithreaded programming primarily in its algorithmic foundation and its ability to deliver performance guarantees. Nevertheless, the algorithmic work builds on earlier work focused on more restrictive models of multithreaded computation. Likewise, the Cilk programming model and runtime system—including Cilk-NOW—build on ideas found in earlier systems. In this section, we look at other theoretical results and systems that address scheduling issues for dynamic parallel computation. We shall not look at data-parallel systems [8, 53] nor at systems focused on infrastructure such as distributed shared memory [4, 6, 29, 39, 59, 60, 66, 73, 87, 92, 93] or message passing [43, 96, 104, 105].

Substantial research has been reported in the theoretical literature concerning the scheduling of dynamic computations. In contrast to our research on multithreaded computations, however, other theoretical research has tended to ignore space requirements and communication costs. Related work in this area includes a randomized work-stealing algorithm for load balancing independent jobs [89]; algorithms for dynamically embedding trees in fixed-connection networks [5, 71]; and algorithms for backtrack search and branch-and-bound [61, 65, 75, 86, 109]. Backtrack search can be viewed as a multithreaded computation with no synchronization, and in the work just cited, the only algorithm with reasonable space bounds is the random work-stealing algorithm of Karp and Zhang [65, 109], though they did not make this observation until the later work of Zhang and Ortynski [108]. Our results specialize to match theirs.

Likewise, most of the systems-oriented work on multithreading has ignored the space issue. Notable exceptions include the $k$-bounded loops of Culler and Arvind [34, 35, 36] and the throttling mechanism of Ruggiero and Sargeant [90]. These techniques and others [56, 57] have met with some success, though none have any algorithmic foundation.

In algorithmic work that considers space requirements or communication costs, most prior work has focused on cases like backtrack search with no synchronization or has focused on time and space to the exclusion of communication costs. Besides the work of Karp, Zhang, and Ortynski already mentioned, Wu and Kung [107] proved a lower bound on communication requirements in parallel divide-and-conquer programs, and Wu [106] gives a distributed algorithm for scheduling parallel divide-and-conquer programs on fixed-connection networks. These analyses focus on the tree-growing stage of divide-and-conquer programs, so they do not consider the case when synchronization is involved. For programs with nested fine-grained parallelism, Blelloch, Gibbons, and Matias [9] give a scheduling algorithm and prove that it is efficient with respect to both time and space. Burton [18] shows how to limit space in certain parallel computations without causing deadlock, and Burton and Simpson [19] give an offline algorithm that is efficient with respect to time and space in a very general model of multithreaded computation. In contrast, our work presents a scheduling algorithm that is distributed and online and is provably efficient with respect to time, space, and communication for a large class of multithreaded computations. Like the algorithm of Karp and Zhang, our algorithm uses the technique of random work stealing.

The work-stealing idea is not a new one, but until our results, studies of work stealing have been based on heuristic notions and the algorithmic work has focused on particularly simple types of computations, such as the backtrack search already discussed. The work-stealing idea dates back at least as far as Burton and Sleep's research [20] on parallel execution of functional programs and Halstead's implementation of Multilisp [51, 52]. These researchers observed that heuristically, by having each processor work as if it is the only one (i.e., in serial, depth-first order) and having idle processors steal threads from others, space requirements and communication requirements should be curbed. Since then, many researchers have implemented variants on this strategy [41, 42, 44, 50, 67, 70, 77, 81, 84, 94, 103]. Cilk's work-stealing scheduler is very similar to the schedulers in some of these other systems, though Cilk's algorithm uses randomness and is provably efficient.

Many multithreaded programming languages and runtime systems are based on heuristic scheduling techniques. Though systems such as Charm [91], COOL [27, 28], Id [3, 37, 80], Olden [22], and others [29, 31, 38, 44, 54, 55, 63, 88, 98] are based on sound heuristics that seem to perform well in practice and generally have wider applicability than Cilk, none are able to provide any sort of performance guarantee or accurate machine-independent performance model. These systems require that performance-minded programmers become intimate with a collection of scheduling heuristics and, in some cases, machine-specific details. In contrast, Cilk delivers guaranteed performance and provides a machine-independent performance abstraction based on work and critical path length.

The use of work and critical path length to analyze parallel algorithms and model application performance is also not new. Work and critical path have been used in the theory community for years to analyze parallel algorithms [64]. Blelloch [8] has developed a performance model for data-parallel computations based on these same two abstract measures. He cites many advantages to such a model over machine-based models. Cilk provides a similar performance model for the domain of multithreaded computation.

Adaptive parallelism, as implemented in Cilk-NOW, finds earlier incarnations, though most parallel runtime systems employ static job-scheduling strategies. Massively parallel supercomputers such as the Cray Research T3D or the Thinking Machines CM5, for example, either dedicate themselves to a single user at a time or gang-timeshare within fixed size partitions [72]. Systems such as Charm [91], the Parform [21], PVM/Hence [96], and others [29, 42, 44, 97] support parallel computing on a network of workstations. In these systems, the set of machines on which the program runs is chosen statically by the user. Distributed operating systems [30, 82, 98, 99] and other systems [32, 40, 68, 74, 79, 110] provide transparent process placement and (in some cases) migration, but these systems are geared towards large serial programs or coarse-grain distributed programs. A system that does provide adaptive parallelism is Piranha [23, 46, 62]. (The creators of Piranha appear to have coined the term "adaptive parallelism.") Based on Linda [24], Piranha's adaptive parallelism leverages structure in the Linda programming model much as Cilk-NOW leverages structure in the Cilk programming model. Adaptive parallelism is also present in the Benevolent Bandit Laboratory (BBL) [40], a PC-based system. The BBL system architecture is closely related to Cilk-NOW's. The Prospero resource manager [78] also employs a similar system architecture. Runtime systems for the programming language COOL [28] running on symmetric multiprocessors [101, 102] and cache-coherent, distributed, shared-memory machines [26, 69] use process control to support adaptive parallelism. These systems rely on special-purpose operating system and hardware support. In contrast, Cilk-NOW supports adaptive parallelism entirely in user-level software on top of commercial hardware and operating systems.

We are currently aware of no other runtime system for a multithreaded programming language that provides transparent fault tolerance.

## 1.3 Contributions of this thesis

This thesis shows that with a well-structured programming model built on an algorithmic foundation, multithreaded programs can be executed efficiently with guaranteed and predictable performance. The principal contributions of this thesis are the following:

- *A lower bound that shows that without some kind of structure, it is possible to write multithreaded programs that are impossible to schedule efficiently for parallel execution.* We give the first proof that parallelism obtained by nonstrict execution order may be chimerical. In these cases, any attempt to exploit this parallelism requires vastly more space per processor than required for a 1-processor execution.

- *An upper bound that shows that strictness is sufficient structure for efficient parallel execution of multithreaded programs.* We show that any parallelism obtainable with strict execution order can be exploited without using any more space per processor than required for a 1-processor execution.

- *A provably efficient scheduling algorithm for multithreaded programs.* This online and distributed algorithm is based on the popular technique of work stealing. We prove that for the case of fully strict computations, this algorithm is simultaneously efficient with respect to space, time, and communication. This is the first provably efficient algorithm for scheduling multithreaded computations with synchronization.

- *Cilk: a language and runtime system for multithreaded programming.* The Cilk language is based on C, and the Cilk runtime system uses the provably efficient work-stealing scheduler. Using several applications, we show empirically that Cilk's work-stealing scheduler is efficient in practice. We further show that the performance of Cilk applications can be predicted accurately using a simple model based on two abstract measures: work and critical path length.

- *Cilk-NOW: an implementation of the Cilk runtime system on a network of workstations.* Cilk-NOW leverages the structure in Cilk's programming model in order to implement adaptive parallelism and fault tolerance. These features allow Cilk programs to run efficiently on a network of workstations. With adaptive parallelism, Cilk programs can run on a set of workstations that grows and shrinks dynamically. With fault tolerance, Cilk programs continue to run even if some of the workstations crash.

The remainder of this thesis is organized as follows. In Chapter 2, we present a graph-theoretic model of multithreaded computation. This model forms the basis for our analysis. In Chapter 3, we present the lower bound showing that in general, there exist multithreaded computations for which efficient scheduling is impossible. We defer the proof of this lower bound to Appendix A. Additionally, in Chapter 3, we define strictness and we show that strictness is a sufficient structure to guarantee the existence of efficient execution schedules. In Chapter 4, we present and analyze the work-stealing scheduling algorithm. The essential ideas of this algorithm are implemented in the runtime system for the Cilk multithreaded language. In Chapter 5, we present the Cilk language and the implementation of the Cilk runtime system. We also show both empirically and analytically that the Cilk runtime system delivers efficient and predictable performance. The analysis builds on the work of Chapter 4. In Chapter 6, we present the implementation of adaptive parallelism and fault tolerance in Cilk-NOW. Finally, in Chapter 7, we conclude and discuss current and planed work to add distributed shared memory to Cilk using "dag consistency" [11]. The reader interested only in the system-building contributions of this thesis may safely skip ahead to Chapter 5 and pass over Section 5.3. Information about the current and forthcoming Cilk software releases can be found in Appendix B.

# Chapter 2

# A model of multithreaded computation

The execution of a multithreaded program grows a directed, acyclic graph of "instructions" and a tree of "threads." The size and longest path length of the graph provide bounds on the achievable execution time. The height of the tree provides a bound on the achievable execution space. In this chapter, we shall introduce our graphical model of multithreaded computation and then use it to derive simple bounds on execution time and space. Section 2.1 presents the model. We then use this model in Section 2.2 to derive time bounds and in Section 2.3 to derive space bounds. This model and associated bounds equip us with an algorithmic foundation for analyzing scheduling algorithms (Chapters 3 and 4) and predicting Cilk application performance (Chapter 5).

## 2.1 Multithreaded computation

A *multithreaded computation* models the time and space resource requirements in the execution of a multithreaded program. This model contains a graph of instructions and a tree of threads that unfold dynamically during program execution. In this section, we present this model and define what it means for a parallel computer to execute a multithreaded computation. In the next two sections we shall quantify and bound the time and space requirements.

A multithreaded computation is composed of a set of *threads*, each of which is a sequential ordering of unit-size *instructions*. A processor takes one unit of time to execute one instruction. In the example computation of Figure 2.1, each shaded block is a thread

The research reported in this chapter is joint work with Charles Leiserson of MIT's Laboratory for Computer Science and was first published in [13].

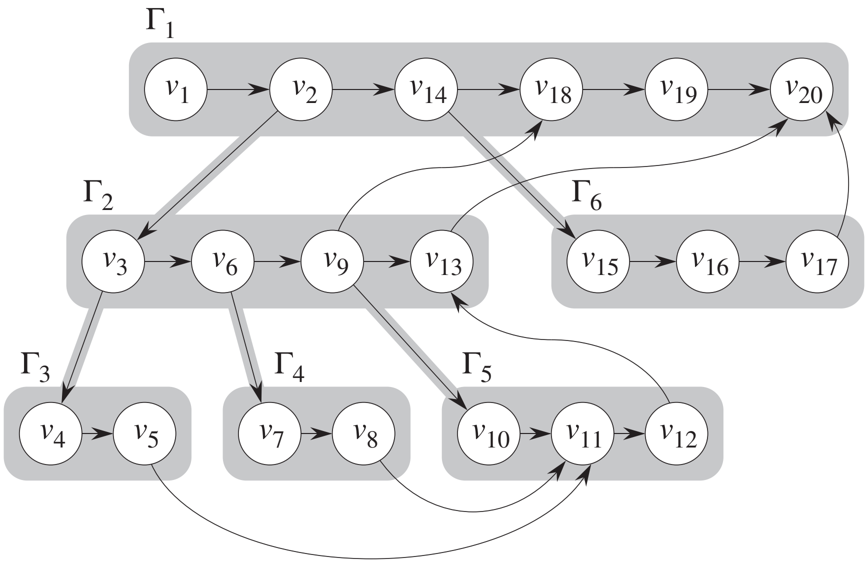



**Figure 2.1**: A multithreaded computation. This computation contains 20 instructions $v_1, v_2, \ldots, v_{20}$ and 6 threads $\Gamma_1, \Gamma_2, \ldots, \Gamma_6$.

with circles representing instructions and the horizontal edges, called *continue* edges, representing the sequential ordering. Thread $\Gamma_5$ of this example contains 3 instructions: $v_{10}$, $v_{11}$, and $v_{12}$. The instructions of a thread must execute in this sequential order from the first (leftmost) instruction to the last (rightmost) instruction. In order to execute a thread, we allocate for it a block of memory, called an *activation frame*, that the instructions of the thread can use to store the values on which they compute.

An *execution schedule* for a multithreaded computation determines which processors of a parallel computer execute which instructions at each step. In any given step of an execution schedule, each processor either executes a single instruction or sits idle. A 3-processor execution schedule for our example computation (Figure 2.1) is shown in Figure 2.2. At step 3 of this example, processors $p_1$ and $p_2$ each execute an instruction while processor $p_3$ sits idle. An execution schedule depends on the particular multithreaded computation, since it must observe the sequential ordering of the instructions in each thread. Specifically, if an instruction has a predecessor—that is, an instruction that connects to it via a continue edge—in its thread, then no processor may execute that instruction until after the predecessor has been executed.

During the course of its execution, a thread may create, or *spawn*, other threads. Spawning a thread is like a subroutine call, except that the spawning thread can operate concurrently with the spawned thread. We consider spawned threads to be children of the thread that did the spawning, and a thread may spawn as many children as it desires. In this way, the threads are organized into a *spawn tree* as indicated in Figure 2.1 by the downward-

| step | living threads | | | | | | processor activity | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | $p_1$ | $p_2$ | $p_3$ |
| 1 | $\mathbf{\Gamma}_1$ | | | | | | $v_1$ | | |
| 2 | $\mathbf{\Gamma}_1$ | | | | | | $v_2$ | | |
| 3 | $\mathbf{\Gamma}_1$ | $\mathbf{\Gamma}_2$ | | | | | $v_3$ | $v_{14}$ | |
| 4 | $\Gamma_1$ | $\mathbf{\Gamma}_2$ | $\mathbf{\Gamma}_3$ | | | $\mathbf{\Gamma}_6$ | $v_4$ | $v_6$ | $v_{15}$ |
| 5 | $\Gamma_1$ | $\mathbf{\Gamma}_2$ | $\Gamma_3$ | $\mathbf{\Gamma}_4$ | | $\mathbf{\Gamma}_6$ | $v_5$ | $v_9$ | $v_{16}$ |
| 6 | $\mathbf{\Gamma}_1$ | $\Gamma_2$ | | $\mathbf{\Gamma}_4$ | $\mathbf{\Gamma}_5$ | $\mathbf{\Gamma}_6$ | $v_7$ | $v_{10}$ | $v_{17}$ |
| 7 | $\mathbf{\Gamma}_1$ | $\Gamma_2$ | | $\mathbf{\Gamma}_4$ | $\Gamma_5$ | | $v_8$ | $v_{18}$ | |
| 8 | $\mathbf{\Gamma}_1$ | $\Gamma_2$ | | | $\mathbf{\Gamma}_5$ | | | $v_{19}$ | $v_{11}$ |
| 9 | $\Gamma_1$ | $\Gamma_2$ | | | $\mathbf{\Gamma}_5$ | | | | $v_{12}$ |
| 10 | $\Gamma_1$ | $\mathbf{\Gamma}_2$ | | | | | | | $v_{13}$ |
| 11 | $\mathbf{\Gamma}_1$ | | | | | | | | $v_{20}$ |

**Figure 2.2**: A 3-processor execution schedule for the computation of Figure 2.1. This schedule lists the living threads at the start of each step, and the instruction (if any) executed by each of the 3 processors, $p_1$, $p_2$, and $p_3$, at each step. Living threads that are ready are listed in bold. The other living threads are stalled.

pointing, shaded edges, called *spawn* edges, that connect threads to their spawned children. The spawn tree is the parallel analog of a call tree. In our example computation, the spawn tree's *root* thread $\Gamma_1$ has two children, $\Gamma_2$ and $\Gamma_6$, and thread $\Gamma_2$ has three children, $\Gamma_3$, $\Gamma_4$, and $\Gamma_5$. Threads $\Gamma_3$, $\Gamma_4$, $\Gamma_5$, and $\Gamma_6$, which have no children, are *leaf* threads.

Each spawn edge goes from a specific instruction—the instruction that actually does the spawn operation—in the parent thread to the first instruction of the child thread. An execution schedule must obey this edge in that no processor may execute an instruction in a spawned child thread until after the spawning instruction in the parent thread has been executed. In our example computation (Figure 2.1), due to the spawn edge $(v_6, v_7)$, instruction $v_7$ cannot be executed until after the spawning instruction $v_6$. Consistent with our unit-time model of instructions, a single instruction may spawn at most one child. When the spawning instruction is executed, we allocate an activation frame for the new child thread. Once a thread has been spawned and its frame has been allocated, we say the thread is *alive* or *living*. When the last instruction of a thread gets executed, the thread *dies* and we deallocate its frame. In our 3-processor execution schedule (Figure 2.2), thread $\Gamma_5$ is spawned at step 5 and dies at step 9. Therefore, it is living at steps 6, 7, 8, and 9.

Spawns introduce parallelism, but a given execution schedule may or may not exploit this parallelism. For example, when thread $\Gamma_2$ spawns children $\Gamma_3$ and $\Gamma_4$, these two child threads may be executed concurrently on different processors. Alternatively, a single processor may execute both threads, possibly interleaving their instructions in some way.

In addition to the continue and spawn edges, a multithreaded computation may also contain dependency edges, as illustrated in Figure 2.1 by the curved edges. Dependency edges model the data and control dependencies between threads. As an example of a data dependency, consider an instruction that produces a data value consumed by another instruction. Such a producer/consumer relationship precludes the consuming instruction from executing until after the producing instruction. A dependency edge from the producing instruction to the consuming instruction enforces this ordering. An execution schedule must obey this edge in that no processor may execute the consuming instruction until after the producing instruction has been executed. For example, due to the dependency edge $(v_5, v_{11})$, instruction $v_{11}$ cannot be executed until after $v_5$.

Dependency edges allow threads to synchronize. Consider again the case of a dependency edge that models a producer/consumer data dependency. If the execution of the consuming thread arrives at the consuming instruction before the producing instruction has executed, then execution of the consuming thread cannot continue—the thread *stalls* and must suspend its execution. Once the producing instruction executes, the dependency is *resolved*, thereby *enabling* the consuming thread to resume with its execution—the thread becomes *ready*. Thus, the dependency edge allows the consuming thread to synchronize with the producing thread. For example, at step 4 of our 3-processor execution schedule (Figure 2.2), thread $\Gamma_1$ is stalled at instruction $v_{18}$, because instruction $v_9$ has not yet been executed. At step 5 instruction $v_9$ is executed by processor $p_2$, thereby enabling thread $\Gamma_1$. At step 6, thread $\Gamma_1$ is ready at instruction $v_{18}$. A multithreaded computation does not model the mechanism by which dependencies get resolved or unresolved dependencies get detected. In Chapter 5 we present Cilk's implementation of such a mechanism.

An execution schedule must obey the constraints given by the dependency, spawn, and continue edges of the computation. These edges form a directed graph of instructions, and no processor may execute an instruction until after all of the instruction's predecessors in this graph have been executed. So that execution schedules exist, this graph must be acyclic. That is, it must be a directed acyclic graph, or *dag*. At any given step of an execution schedule, an instruction is *ready* if all of its predecessors in the dag have been executed. Only ready instructions may be executed.

The notion of an execution schedule is independent of any real machine characteristics. An execution schedule simply requires that no processor executes more than one instruction per time step and every instruction is executed at a time step after all of its predecessor instructions (which connect to it via continue, spawn, or dependency edges) have been executed. A given execution schedule may not be viable for a real machine, since the schedule may not account for properties such as communication latency. For example, in our 3-

processor execution schedule (Figure 2.2), instruction $v_{11}$ is executed at step 8 by processor $p_3$ exactly one step after $v_8$ is executed by processor $p_1$, even though there is a dependency between them that surely requires some latency to be resolved. In later chapters we will turn attention to computing execution schedules for real machines.

To summarize, a multithreaded computation can be viewed as a dag of instructions connected by continue, spawn, and dependency edges. The instructions are connected by continue edges into threads, and the threads form a tree with the spawn edges. When a thread is spawned, an activation frame is allocated and this frame remains allocated as long as the thread remains alive. A living thread may be either ready or stalled due to an unresolved dependency.

It is important to note the difference between what we are calling a multithreaded computation and a program. A multithreaded computation is the "parallel instruction stream" resulting from the execution of a multithreaded program with a given set of inputs. Unlike a serial computation in which the instruction stream is totally ordered, a multithreaded computation only partially orders its instructions. In general, a multithreaded computation is not a statically determined object, rather the computation unfolds dynamically during execution as determined by the program and the input data. For example, a program may have conditionals, and therefore, the order of instructions (or even the set of instructions) executed in a thread may not be known until the thread is actually executed. We can think of a multithreaded computation as encapsulating both the program and the input data. The computation then reveals itself dynamically during execution.

We shall characterize the time and space of an execution of a multithreaded computation in terms of three fundamental parameters: work, critical path length, and stack depth. We first introduce work and critical path length, which relate to the execution time, and then we focus on stack depth, which relates to the storage requirements.

## 2.2 Execution time

Execution time requirements are captured in two fundamental measures of the multithreaded computation's graph of instructions: work and critical path length. In this section, we define the work and critical path length of a multithreaded computation, and we use these measures to derive simple bounds on execution time.

If we ignore the shading in Figure 2.1 that organizes instructions into threads, our multithreaded computation is just a dag of instructions. The dag corresponding to the example

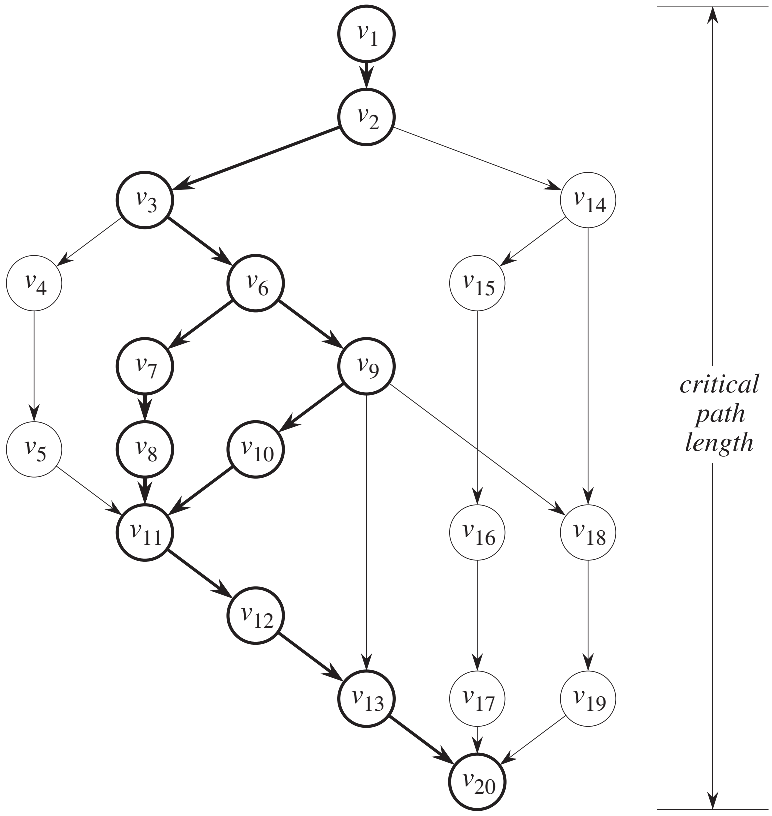



**Figure 2.3**: The dag corresponding to the computation of Figure 2.1. The critical paths, each 10 instructions long, are shown bold.

computation of Figure 2.1 is shown in Figure 2.3. We define the *work* of the computation to be the total number of instructions and the *critical path length* to be the length of a longest directed path in the dag. In the case of our example, the work is 20 instructions and the critical path length is 10 instructions.

We quantify and bound the execution time of a computation in terms of the computation's work and critical path length. For a given computation, let $T(\mathcal{X})$ denote the time to execute the computation using a given $P$-processor execution schedule $\mathcal{X}$, and let

$$T_P = \min_{\mathcal{X}} T(\mathcal{X})$$

denote the minimum execution time over all $P$-processor execution schedules $\mathcal{X}$. Then $T_1$ is the work of the computation, since a 1-processor computer can only execute one instruction at each step, and $T_\infty$ is the critical path length, since even with arbitrarily many processors, each instruction on a path must execute serially. In our example dag (Figure 2.3), $T_1 = 20$ (a single processor can execute the instructions in the order $v_1, v_2, \ldots, v_{20}$ since this is a topological sort of the dag), and $T_\infty = 10$ (an infinite-processor execution schedule that achieves

| step | living threads | | | | | | processor activity | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ | $p_6 \cdots$ |
| 1 | $\mathbf{\Gamma}_1$ | | | | | | $v_1$ | | | | | |
| 2 | $\mathbf{\Gamma}_1$ | | | | | | $v_2$ | | | | | |
| 3 | $\mathbf{\Gamma}_1$ | $\mathbf{\Gamma}_2$ | | | | | $v_3$ | | | $v_{14}$ | | |
| 4 | $\Gamma_1$ | $\mathbf{\Gamma}_2$ | $\mathbf{\Gamma}_3$ | | | $\mathbf{\Gamma}_6$ | $v_4$ | $v_6$ | | $v_{15}$ | | |
| 5 | $\Gamma_1$ | $\mathbf{\Gamma}_2$ | $\mathbf{\Gamma}_3$ | $\mathbf{\Gamma}_4$ | | $\mathbf{\Gamma}_6$ | $v_5$ | $v_7$ | $v_9$ | $v_{16}$ | | |
| 6 | $\mathbf{\Gamma}_1$ | $\Gamma_2$ | | $\mathbf{\Gamma}_4$ | $\mathbf{\Gamma}_5$ | $\mathbf{\Gamma}_6$ | | $v_8$ | $v_{10}$ | $v_{17}$ | $v_{18}$ | |
| 7 | $\mathbf{\Gamma}_1$ | $\Gamma_2$ | | | $\mathbf{\Gamma}_5$ | | | $v_{11}$ | | | $v_{19}$ | |
| 8 | $\Gamma_1$ | $\Gamma_2$ | | | $\mathbf{\Gamma}_5$ | | | $v_{12}$ | | | | |
| 9 | $\Gamma_1$ | $\mathbf{\Gamma}_2$ | | | | | | $v_{13}$ | | | | |
| 10 | $\mathbf{\Gamma}_1$ | | | | | | | $v_{20}$ | | | | |

**Figure 2.4**: An infinite-processor execution schedule for the dag of Figure 2.3. The maximum number of instructions executed at any time step is 4 (steps 5 and 6), and the average number of instructions executed per time step is 2 (20 total instructions divided by 10 steps).

this time is shown in Figure 2.4).

The work $T_1$ and critical path length $T_\infty$ are not intended to denote the execution time on any real single-processor or infinite-processor machine. These quantities are abstractions of a computation and are independent of any real machine characteristics such as communication latency. We can think of $T_1$ and $T_\infty$ as execution times on an ideal machine with no scheduling overhead. Nevertheless, we show in Chapter 5 that despite their abstract nature, with a good scheduling algorithm, work and critical path length have a lot to say about actual execution time of actual programs on actual machines.

Still viewing the computation as a dag, we borrow some basic results on dag scheduling to bound $T_P$. A computer with $P$ processors can execute at most $P$ instructions per step, and since the computation has $T_1$ instructions, we have the lower bound $T_P \geq T_1/P$. And, of course, we also have the lower bound $T_P \geq T_\infty$. Early independent work by Brent [16, Lemma 2] and Graham [48, 49] yields the upper bound $T_P \leq T_1/P + T_\infty$. The following theorem extends these results minimally to show that this upper bound on $T_P$ can be obtained by any *greedy schedule*: one in which at each step of the execution, if at least $P$ instructions are ready, then $P$ instructions execute, and if fewer than $P$ instructions are ready, then all execute. Both of our example schedules (Figures 2.2 and 2.4) are greedy.

**Theorem 2.1 (The greedy-scheduling theorem)** *For any multithreaded computation with work $T_1$ and critical path length $T_\infty$ and for any number $P$ of processors, any greedy $P$-processor execution schedule $\mathcal{X}$ achieves $T(\mathcal{X}) \leq T_1/P + T_\infty$.*

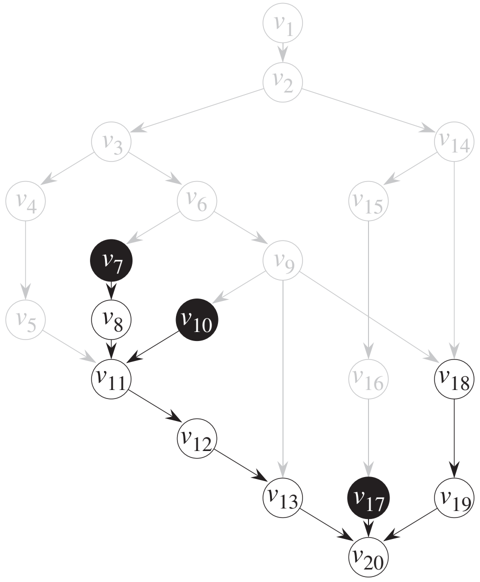



**Figure 2.5**: Step 6 in the 3-processor execution of our example computation. The faint instructions have already been executed. The white-on-black instructions are the ones actually executed at this step. All 3 processors do work at this step.

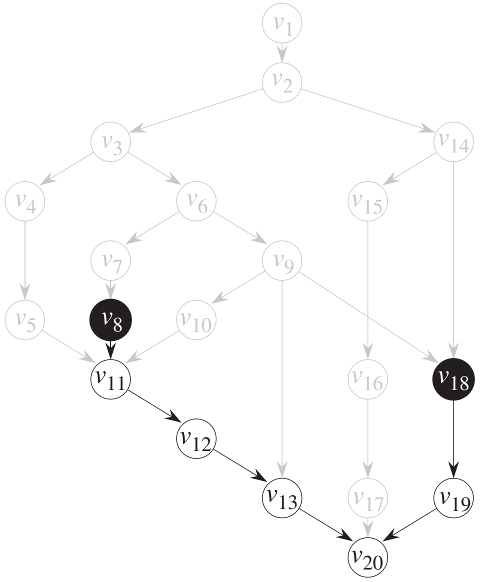



**Figure 2.6**: Step 7 in the 3-processor execution of our example computation. At this step a processor sits idle. Nevertheless, every instruction with in-degree 0 get executed, and consequently, the length of the critical path is reduced by 1.

*Proof:* Let $G = (V, E)$ denote the underlying dag of the computation. Thus, we have $|V| = T_1$, and a longest directed path in $G$ has length $T_\infty$. Consider a greedy $P$-processor execution schedule $\mathcal{X}$, where the set of instructions executed at time step $t$, for $t = 1, 2, \ldots, k$, is denoted $\mathcal{V}_t$, with $k = T(\mathcal{X})$. The $\mathcal{V}_t$ form a partition of $V$.

We shall consider the progression $\langle G_0, G_1, G_2, \ldots, G_k \rangle$ of dags, where $G_0 = G$, and for $t = 1, 2, \ldots, k$, we have $V_t = V_{t-1} - \mathcal{V}_t$ and $G_t$ is the subgraph of $G_{t-1}$ induced by $V_t$. In other words, $G_t$ is obtained from $G_{t-1}$ by removing from $G_{t-1}$ all the instructions that are executed by $\mathcal{X}$ at step $t$ and all edges incident on these instructions. We shall show that each step of the execution either decreases the size of the dag or decreases the length of the longest path in the dag.

We account for each step $t$ according to $|\mathcal{V}_t|$. Consider a step $t$ with $|\mathcal{V}_t| = P$. (Such a step of our example 3-processor execution (Figure 2.2) is shown in Figure 2.5.) In this case, $|V_t| = |V_{t-1}| - P$, so since $|V| = T_1$, there can be at most $\lfloor T_1/P \rfloor$ such steps. Now consider a step $t$ with $|\mathcal{V}_t| < P$. (Such a step of our 3-processor execution is shown in Figure 2.6.) In this case, since $\mathcal{X}$ is greedy, $\mathcal{V}_t$ must contain every vertex of $G_{t-1}$ with in-degree 0. Therefore, the length of a longest path in $G_t$ is one less than the length of a longest path in $G_{t-1}$. Since the length of a longest path in $G$ is $T_\infty$, there can be no more than $T_\infty$ steps $t$ with $|\mathcal{V}_t| < P$.

Consequently, the time it takes the $P$-processor schedule $\mathcal{X}$ to execute the computation is $T(\mathcal{X}) \le \lfloor T_1/P \rfloor + T_\infty \le T_1/P + T_\infty$. ■

The greedy-scheduling theorem (Theorem 2.1) can be interpreted in two important ways. First, the time bound given by the theorem says that any greedy schedule yields an execution time that is within a factor of 2 of an optimal schedule, which follows because $T_1/P + T_\infty \le 2\max\{T_1/P, T_\infty\}$ and $T_P \ge \max\{T_1/P, T_\infty\}$. This observation was first made by Graham [48]. Second, the greedy-scheduling theorem tells us when we can obtain *linear speedup*, that is, when we can find a $P$-processor execution schedule $\mathcal{X}$ such that $T(\mathcal{X}) = \Theta(T_1/P)$. Specifically, when the number $P$ of processors is no more than $T_1/T_\infty$, then $T_1/P \ge T_\infty$, which implies that for a greedy schedule $\mathcal{X}$, we have $T(\mathcal{X}) \le 2T_1/P$. The ratio $T_1/T_\infty$ is called the *average parallelism*. Looking at the example dag of Figure 2.3, if we think of the work $T_1$ as the area of the dag and the critical path length $T_\infty$ as the height of the dag, then the average parallelism $T_1/T_\infty$ is the average width of the dag. Our example has average parallelism $T_1/T_\infty = 20/10 = 2$. We can also think of the average parallelism as the average number of instructions executed per step of a greedy, infinite-processor execution schedule (Figure 2.4). We shall be especially interested in the regime where $P = O(T_1/T_\infty)$ and linear speedup is possible, since outside this regime, linear speedup is impossible to achieve because $T_P \ge T_\infty$.

These results on dag scheduling have been known for years. A multithreaded computation, however, adds further structure to the dag: the partitioning of instructions into a tree of threads. This additional structure allows us to quantify the space required in executing a multithreaded computation. Once we have quantified space requirements, we will look back at the greedy-scheduling theorem and consider whether there exist execution schedules that achieve similar time bounds while also making efficient use of space. Of course, we will have to quantify a space bound to capture what we mean by "efficient use of space."

## 2.3 Execution space

A multithreaded computation models execution space requirements in the spawn tree of threads. In this section, we shall focus on a single measure of this tree: stack depth. We use this measure to derive simple bounds on execution space, and then we formalize our goal to achieve "efficient use of space."

In our analysis of space requirements, we shall only analyze "stack-like" memory. We say that memory is *stack-like* if its allocation and deallocation obey the following rules. No instruction allocates memory if its thread could possibly have a living child when the instruction is executed, and all memory allocated for a thread is deallocated before or when the thread dies. For now, we shall simplify our notion of stack-like memory by assuming that all memory for a thread is allocated in an activation frame when the thread is spawned and deallocated when the thread dies. With this assumption, the space being used at any time step $t$ is equal to the sum of the sizes of the activation frames of the threads that are living at step $t$, and the total space used in executing a computation is the maximum such value over the course of the execution. For now, we shall also assume that a parent thread remains alive until all its children die. Though these assumptions are not absolutely necessary, they simplify our analysis of space utilization by giving the execution a natural structure. In the next chapter, we shall see how these assumptions can be relaxed to account for arbitrary stack-like memory.

The multithreaded computation's spawn tree of threads naturally gives rise to a corresponding tree of activation frames. We call this tree the *activation tree*. The activation tree corresponding to the example computation of Figure 2.1 is shown in Figure 2.7. Each node of the activation tree is an activation frame drawn as a block with height equal to the size of the frame. We define the *stack depth* of a thread to be the sum of the sizes of the activation frames of all its ancestors, including itself. The *stack depth* of a multithreaded computation is the maximum stack depth of any thread. In Figure 2.7, each child frame has its top aligned with its parent's bottom. Thus, we can view the computation's stack depth as the depth of the activation tree.

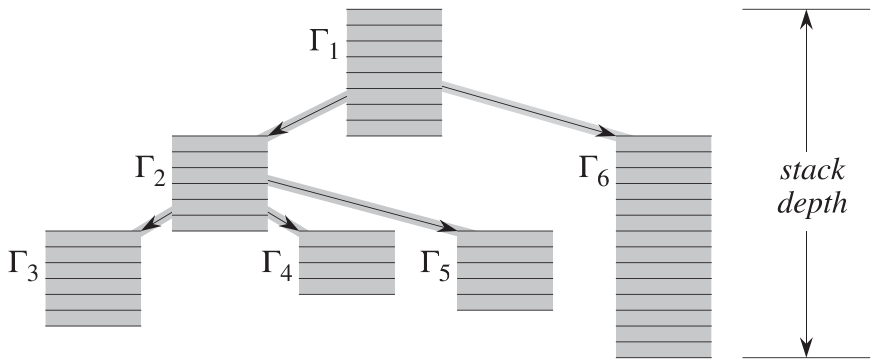



**Figure 2.7**: The activation tree corresponding to the computation of Figure 2.1.

We define the *spawn subtree* at any time step $t$ to be the portion of the spawn tree consisting of just those threads that are alive at step $t$. Analogously, we define the *activation subtree*. The activation subtree at step 7 of our example 3-processor execution (Figure 2.2) is shown in Figure 2.8. The space used at time step $t$ equals the size of the activation subtree at step $t$, and the total space used in executing a computation is the maximum such value over the course of the execution. In our example 3-processor execution, this maximum space usage occurs at step 5 when the activation subtree contains a frame for every thread except $\Gamma_5$.

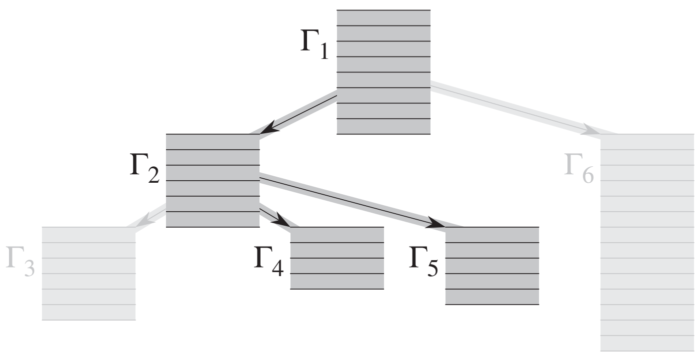



**Figure 2.8**: The activation subtree at step 7 in the 3-processor execution of our example computation. Threads not currently living are faint.

We shall denote the space required by a $P$-processor execution schedule $\mathcal{X}$ of a multithreaded computation by $S(\mathcal{X})$. Since we can always simulate a $P$-processor execution with a 1-processor execution that uses no more space, we have $S_1 \leq S(\mathcal{X})$, where

$$S_1 = \min_{\mathcal{X}} S(\mathcal{X})$$

denotes the minimum space used over all 1-processor execution schedules $\mathcal{X}$.

The following simple theorem shows that the stack depth of a computation is a lower bound on the space required to execute it.

**Theorem 2.2** *Let $\mathcal{S}$ be the stack depth of a multithreaded computation, and let $\mathcal{X}$ be a P-processor execution schedule of the computation. Then, we have $S(\mathcal{X}) \geq \mathcal{S}$, and more specifically, we have $S_1 \geq \mathcal{S}$.*

*Proof:* In any schedule, the leaf thread with greatest stack depth must be alive at some time step. Since we assume that if a thread is alive, its parent is alive, when the deepest leaf thread is alive, all its ancestors are alive, and hence, all its ancestors' frames are allocated. But, the sum of the sizes of its ancestors' activation frames is just the stack depth. Since $S(\mathcal{X}) \geq \mathcal{S}$ holds for any $P$-processor schedule $\mathcal{X}$ and all $P$, it holds for the minimum-space execution schedule, and hence, we have $S_1 \geq \mathcal{S}$. ■

Given the lower bound of stack depth on the space used by a $P$-processor schedule, it is natural to ask whether the stack depth can be achieved as an upper bound. In general, the answer is no, since all the threads in a computation may contain a cycle of dependencies that force all of them to be simultaneously living in any execution schedule. Figure 2.9 illustrates such a computation. For the class of "depth-first" computations, however, space equal to the stack depth can be achieved by a 1-processor schedule.

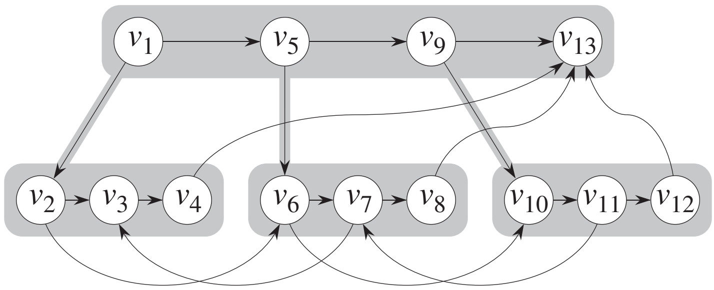



**Figure 2.9**: The dependency edges in this computation form cycles through all of the child threads, though they do not create any cycle of instructions. Any execution schedule must have all 4 of this computation's threads simultaneously living at some step. This computation is not depth-first, since the dependency edges, $(v_7, v_3)$ and $(v_{11}, v_7)$, violate the left-to-right depth-first order, $v_1, v_2, \ldots, v_{13}$.

A *depth-first* multithreaded computation is a multithreaded computation in which the "left-to-right depth-first" search of instructions in the computation always visits all of any given instruction's predecessors in the dag before it visits the given instruction. Specifically, we define the *left-to-right depth-first* order of instructions as follows. If we ignore the dependency edges in the dag and just look at the instructions connected by continue and spawn edges, we have a binary tree rooted at the first instruction of the root thread. Any instruction with 2 children in this tree must connect to one child by a spawn edge and the other child by a continue edge. We consider a child connected to its parent by a spawn edge to be the left child and a child connected to its parent by a continue edge to be the right child. The left-to-right depth-first order is then the order of instructions visited by a preorder walk of this tree [33]. For example, the left-to-right depth-first order for the computation of Figure 2.9 is $v_1, v_2, \ldots, v_{13}$. Now considering the dependency edges again, a multithreaded computation is depth-first if this left-to-right depth-first order yields a 1-processor execution schedule. In other words, the computation is depth-first if none of its dependency edges violate its left-to-right depth-first order. The computation of Figure 2.9 is not depth-first, since the dependency edges, $(v_7, v_3)$ and $(v_{11}, v_7)$, violate the left-to-right depth-first order, $v_1, v_2, \ldots, v_{13}$. On the other hand, our example computation of Figure 2.1 is depth-first: 1 processor executing instructions in the order $v_1, v_2, \ldots, v_{20}$ is a valid execution schedule.

In fact, for depth-first computations, this left-to-right depth-first order produces a 1-processor execution schedule which is just the familiar serial stack-based execution. This execution begins with the root thread and executes its instructions until it either spawns a child thread or dies. If the thread spawns a child, the parent thread is suspended to be resumed only after the child thread dies; the scheduler then begins work on the child, executing the child until it either spawns or dies.

**Theorem 2.3** *For any depth-first multithreaded computation with stack depth $\mathcal{S}$, we have $S_1 = \mathcal{S}$.*

*Proof:* At any time in a serial depth-first execution of the computation, the set of living threads always forms a path from the root. Therefore, the space required is just the stack depth of the computation. By Theorem 2.2, we have $S_1 \geq \mathcal{S}$, and thus the space used is the minimum possible. ∎

For the remainder of this thesis, we shall focus on depth-first multithreaded computations, and therefore, we shall use $S_1$ to denote stack depth.

We now turn our attention to determining how much space $S(\mathcal{X})$ a $P$-processor execution schedule $\mathcal{X}$ can use and still be considered efficient with respect to space usage. Our strategy is to compare the space used by a $P$-processor schedule with the space $S_1$ required by an optimal 1-processor schedule. Of course, we can always ignore $P-1$ of the processors to match the single-processor space bounds, and therefore, our goal is to use small space while obtaining linear speedup. We argue that a $P$-processor execution schedule $\mathcal{X}$ that uses space $S(\mathcal{X}) = O(S_1 P)$ is efficient.

There exist very simple multithreaded computations that require $\Theta(S_1P)$ space in order to achieve linear speedup. Consider the following computation. The root thread is a loop that spawns a child thread for each iteration, and each child thread is a leaf. The computation has the activation tree shown in Figure 2.10. The root activation frame is negligible in size compared with the leaves (its children). A single processor executing this computation uses only the space needed for a single iteration (plus the space used by the root), since upon completion of an iteration, all the memory can be freed and then reused for the next iteration. Thus, $S_1$ is slightly larger than the size of a single leaf activation frame. In general, with $P$ processors, obtaining linear speedup requires executing $\Theta(P)$ iterations concurrently. Such a $P$-processor execution schedule $\mathcal{X}$ has $\Theta(P)$ leaf threads living at some step, and therefore, it uses space $S(\mathcal{X}) = \Theta(S_1P)$.

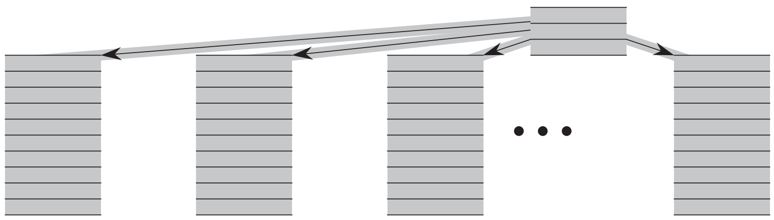


**Figure 2.10**: The activation tree for a multithreaded computation that requires linear expansion of space in order to achieve linear speedup.

A $P$-processor execution schedule $\mathcal{X}$ for which $S(\mathcal{X}) = \Theta(S_1P)$ is said to exhibit *linear expansion of space*. "Reasonable" example computations such as the one just considered show that for some computations, obtaining linear speedup requires a linear expansion of space. For other computations we would like to do better. Nevertheless, a $P$-processor schedule $\mathcal{X}$ for which $S(\mathcal{X}) = O(S_1P)$ is arguably efficient, since on average, each of the $P$ processors needs no more memory than is used by the optimal 1-processor execution.

Recalling that any greedy execution schedule achieves linear speedup (provided that the number of processors is at most proportional to the average parallelism), searching for execution schedules that simultaneously achieve linear speedup and linear expansion of space appears as a reasonable and maybe even modest endeavor. In the next chapter, however, we shall show that for some multithreaded computations—even depth-first multithreaded computations—this search must fail. On the other hand, we also show in the next chapter that for the class of strict (a subclass of depth-first) multithreaded computations, such efficient execution schedules do exist and they are easy to find.

# Chapter 3

# Strict multithreaded computations

Does every depth-first multithreaded computation have an execution schedule that is simultaneously efficient with respect to time and space? And if not, what necessary and sufficient conditions can we place on computations to guarantee the existence of such a schedule? In this chapter, we show that the answer to the first of these questions is no, and we partially answer the second. In Section 3.1, we present the lower bound that answers the first question. Then in Section 3.2, we define a condition called "strictness," and in Section 3.3, we show that strictness is a sufficient condition to guarantee the existence of efficient execution schedules. We leave necessary conditions as an open question. Strictness provides important structure to computations that we leverage in a provably efficient scheduling algorithm (Chapter 4) employed by the Cilk runtime system (Chapter 5) and in the implementation of adaptive parallelism and fault tolerance on a network of workstations (Chapter 6).

## 3.1 A lower bound for the general case

In a study of resource requirements for dataflow programs published in 1988, Culler and Arvind [36] observed applications with parallelism that they conjectured to be "useless." *Useless parallelism* is parallelism that requires excessive amounts of space resource to exploit. While Culler and Arvind argued convincingly that the observed useless parallelism is in fact useless, they came short of a proof, and they left open the possibility of a clever scheduler that might be able to exploit this parallelism without using excessive amounts of space. With "loop-bounding" [34, 35, 36] techniques, they were able to eliminate the useless parallelism with only a small decrease in the average parallelism. Their applications

Some of the research reported in this chapter is joint work with Charles Leiserson of MIT's Laboratory for Computer Science and was first published in [13] and [14].

had only small amounts of useless parallelism.

In this section we show that multithreaded computations may contain vast quantities of *provably* useless parallelism. In particular, we show that there exist depth-first multithreaded computations with large amounts of average parallelism such that any execution schedule attempting to exploit this parallelism must use excessive amounts of space—that is, much more space per processor than required by a 1-processor execution. This lower bound motivates our consideration, in later sections, of computations with more structure, namely, strict multithreaded computations.

Before going on to a more formal statement of the lower bound, it is worth noting the difference between useless and "excess" parallelism. If we have a computation with average parallelism equal to one million and we are executing this computation on a parallel computer with two processors, then the computation has *excess parallelism*. Excess parallelism is not necessarily useless. If our computer actually had one million processors, we might be able to exploit all of the parallelism while using only modest amounts of space per processor.

For any amount of serial space $S_1$ and any (reasonably large) serial execution time $T_1$, we can exhibit a depth-first multithreaded computation with work $T_1$ and stack depth $S_1$ but with provably bad time/space tradeoff characteristics. Being depth-first, we know from Theorem 2.3 (page 25) that our computation can be executed using 1-processor space $S_1$. Furthermore, we know from the greedy-scheduling theorem (Theorem 2.1, page 19) that for any number $P$ of processors, any greedy $P$-processor execution schedule $\mathcal{X}$ achieves $T(\mathcal{X}) \leq T_1/P + T_\infty$. Our computation has critical path length $T_\infty \approx \sqrt{T_1}$, and consequently, for $P = O(\sqrt{T_1})$, a greedy schedule $\mathcal{X}$ yields $T(\mathcal{X}) = O(T_1/P)$—linear speedup. We show, however, that any $P$-processor execution schedule $\mathcal{X}$ achieving $T(\mathcal{X}) = O(T_1/P)$ must use space $S(\mathcal{X}) = \Omega(\sqrt{T_1}(P-1))$. Of course, $\sqrt{T_1}$ may be much larger than $S_1$, and hence, this space bound is nowhere near linear in its space expansion. A proof of the following theorem is presented in Appendix A.

**Theorem 3.1** *For any $S_1 \geq 4$ and any $T_1 \geq 16S_1^2$, there exists a depth-first multithreaded computation with work $T_1$, average parallelism $T_1/T_\infty \geq \sqrt{T_1}/8$, and stack depth $S_1$ such that the following holds. For any number $P$ of processors and any value $\rho$ in the range $1 \leq \rho \leq \frac{1}{8}T_1/T_\infty$, if $\mathcal{X}$ is a $P$-processor execution schedule that achieves speedup $\rho$—that is, $T(\mathcal{X}) \leq T_1/\rho$—then $S(\mathcal{X}) \geq \frac{1}{4}(\rho-1)\sqrt{T_1} + S_1$.* ■

A word about units. Space, $S_1$ and $S(\mathcal{X})$, is measured in bytes. Time, $T_1$, $T_\infty$, and $T(\mathcal{X})$, is measured in microseconds. The inequalities relating space to time all carry constants with appropriate conversion units.

To see what this theorem is saying, consider two possible values of $\rho$. Think of the work $T_1$ as a very large value so that the average parallelism $T_1/T_\infty \geq T_1/(8\sqrt{T_1}) = \Omega(\sqrt{T_1})$ is also reasonably large. Achieving a speedup of 2 should be easy. But if we plug $\rho = 2$ into our bound, we get $S(\mathcal{X}) \geq \sqrt{T_1}/4 + S_1$ for any execution schedule $\mathcal{X}$. Thus, with 2 processors we have the space-per-processor growing proportional to $\sqrt{T_1}$ despite the fact the $S_1$ may be a small constant. And suppose we want to realize speedup out of all of the $\sqrt{T_1}$ parallelism. Plugging $\rho = \Omega(\sqrt{T_1})$ into our lower bound, we get $S(\mathcal{X}) = \Omega(T_1)$. With $\sqrt{T_1}$ processors, we again have the space-per-processor growing proportional to $\sqrt{T_1}$.

In more recent work, Frigo, using a somewhat more general model of multithreaded computation coupled with the same technique as used in the proof of Theorem 3.1, has obtained the following stronger result [45].

**Theorem 3.2 (Frigo)** *For any sufficiently large $T_1$ and any $\rho \geq 2$, there exists a multithreaded computation with work $T_1$ and $1$-processor space requirement $S_1 = O(1)$ such that any execution schedule $\mathcal{X}$ that achieves speedup $\rho$ must use space $S(\mathcal{X}) = \Omega(T_1)$.* ■

With Frigo's construction, just realizing a speedup of 2 requires space-per-processor that grows linearly with the work.

We prove this lower bound by constructing a depth-first multithreaded computation with the desired properties and then proving that it has no efficient execution schedule. Though we defer the proof of Theorem 3.1 to Appendix A, the idea embodied in the constructed computation is as follows. Each processor working on a thread of the computation spawns several child threads all of which subsequently stall on unresolved dependencies. In order to achieve speedup, a scheduler cannot allow all of these processors to sit idle, so some of them must find other ready threads to work on. Again, these threads spawn several children and then they all stall. This process repeats. Thus, to realize any speedup, the scheduler must allow processors to continually spawn new threads even though these newly spawned threads quickly stall still holding their space resources.

A good scheduler should not allocate resources until it is ready to utilize those resources, and a multithreaded computation should not be structured in such a way as to force a scheduler into such an overcommitted situation in order to achieve speedup.

## 3.2 The strictness condition

A strict multithreaded computation contains dependency edges that, as we shall see, forbid the allocation of resources until those resources can be utilized. Specifically, in a *strict* multithreaded computation, every dependency edge goes from a thread to one of its ancestor threads. For example, the computation shown in Figure 3.1(a) is nonstrict, since the bold dependencies, which we refer to as nonstrict dependencies, violate the strictness condition. But by replacing these nonstrict dependencies with new strict ones, we obtain the strict computation shown in Figure 3.1(b).

This strictness condition has the following interpretation in terms of argument passing in functional programs. For any thread $\Gamma$, we define the *subcomputation* rooted at $\Gamma$ as $\Gamma$ and all of its descendant threads. See Figure 3.2. We think of the subcomputation as a function invocation. For any subcomputation, we can think of each edge from a thread outside the subcomputation to a thread inside the subcomputation as passing *arguments* to the function invocation. In the case of a strict computation, each subcomputation has exactly one such edge—the spawn edge from its parent. Thus, in a strict computation, no function can be invoked until all of its arguments have been evaluated, although the arguments can be evaluated in parallel. In other words, a strict computation requires a strict evaluation order (as opposed to a lenient evaluation order) [100].

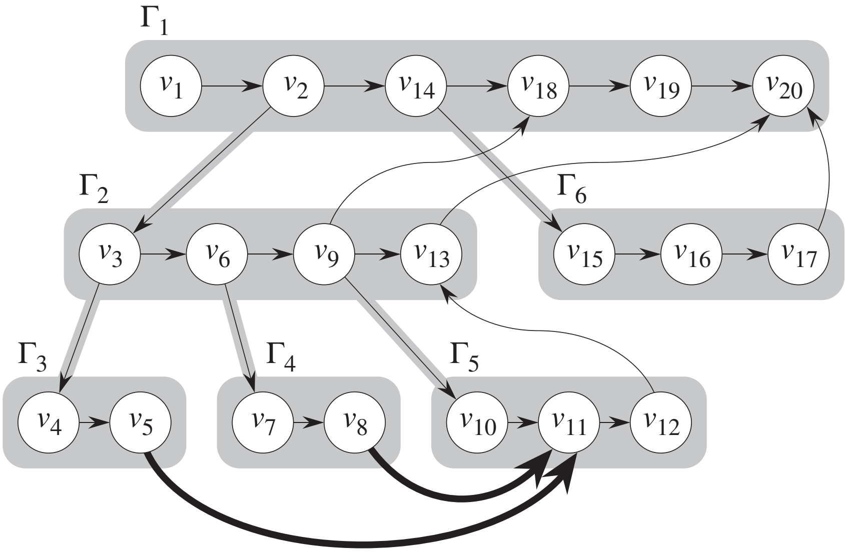
In later chapters we will also consider *fully strict* multithreaded computations. In a fully strict computation, every dependency goes from a thread to its parent. The strict computation in Figure 3.1(b) is also fully strict. Fully strict computations are "well-structured" in that all dependencies from a subcomputation emanate from the subcomputation's root thread.

In the remainder of this section, we show that strict computations are depth-first and that any depth-first computation can be made strict (though possibly at a huge cost in lost parallelism). We begin by showing that strict computations are depth-first.

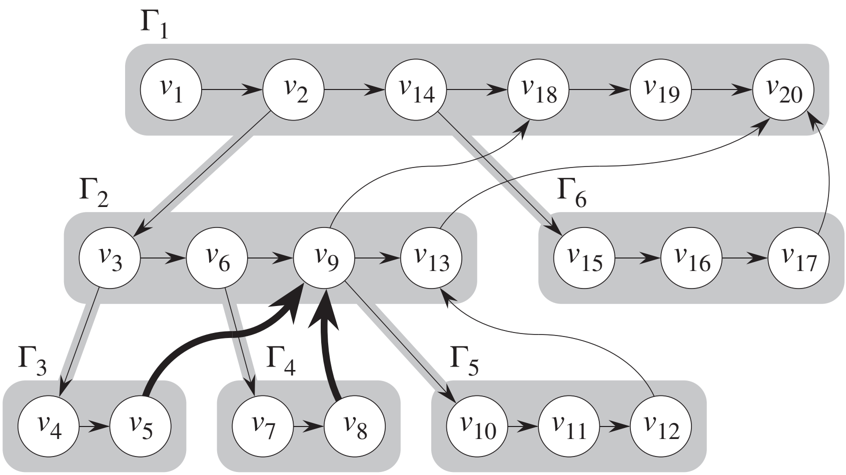
**Theorem 3.3** *Every strict multithreaded computation is a depth-first computation.*

*Proof:* Consider any strict multithreaded computation and let $v_1, v_2, \ldots, v_n$ (with $n = T_1$) denote the left-to-right, depth-first ordering of the instructions. To prove that the computation is depth-first, we must show that every dependency edge is consistent with this ordering. In other words, we must show that for every dependency edge $(v_j, v_k)$, we have $j < k$.

Consider such an edge, and let $\Gamma$ denote the thread containing instruction $v_k$. See Figure 3.3. The strictness conditions says that the thread containing $v_j$ must be a descendant of $\Gamma$, so let $v_i$ be the instruction of $\Gamma$ that spawns the subcomputation containing $v_j$. Observe that the left-to-right depth-first order numbers every instruction in this subcomputation less than every instruction to the right of $v_i$ in $\Gamma$. Thus, since $v_j$ is in this subcomputation, we only need to show that $v_k$ is to the right of $v_i$ in $\Gamma$.




**(a)** Nonstrict




**(b)** Strict

**Figure 3.1**: **(a)** This multithreaded computation (the same as Figure 2.1 on page 14) is nonstrict since it has nonstrict dependencies, shown bold, that go to non-ancestor threads. **(b)** If we replace the nonstrict dependencies with new strict ones, shown bold, we obtain a strict computation since all dependencies go from a child thread to an ancestor thread.

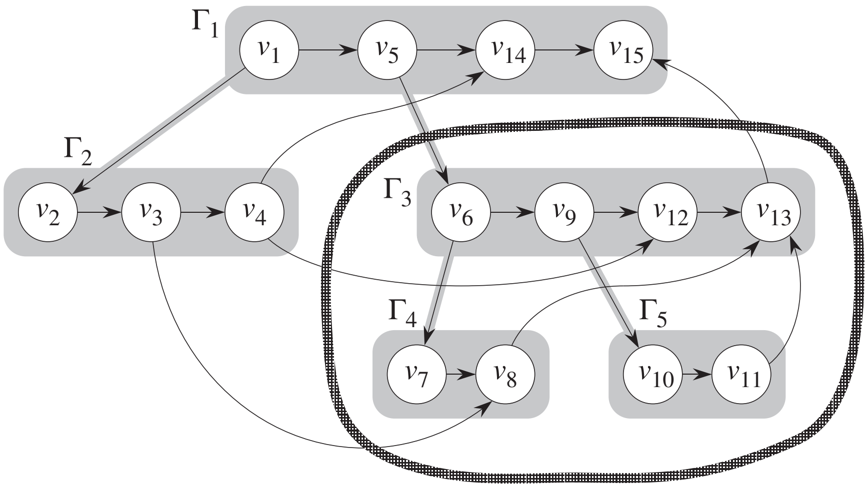



**Figure 3.2**: The spawn edge $(v_5, v_6)$ and the dependency edges, $(v_4, v_{12})$ and $(v_3, v_8)$, pass arguments into the subcomputation rooted at thread $\Gamma_3$. This computation is not strict.

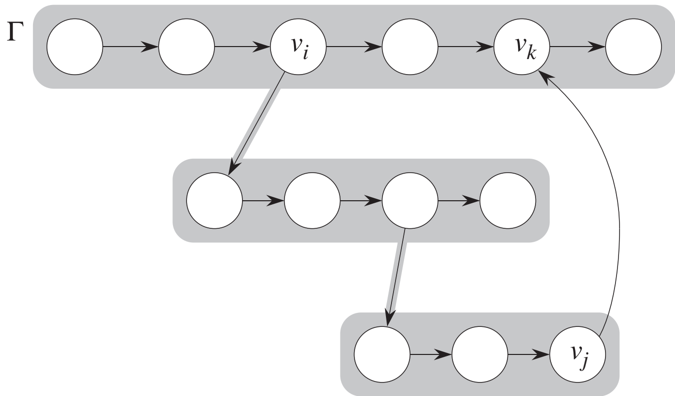



**Figure 3.3**: Every strict computation is depth-first since a strict dependency edge such as $(v_j, v_k)$ must go to an instruction after instruction $v_i$ in $\Gamma$ to avoid introducing a cycle.

Now observe that there is a path from every instruction in $\Gamma$ to the left of (and including) $v_i$ to $v_k$: the path follows continue edges in $\Gamma$ to $v_i$; then it follows spawn and continue edges down to $v_j$; and then it follows our dependency edge $(v_j, v_k)$. Thus, to prevent a cycle, $v_k$ must be to the right of $v_i$. ■

The transformation, illustrated in Figure 3.1, from a depth-first multithreaded computation to a strict multithreaded computation is called *strictifying*, and we say that the resulting computation is the *strictification* of the original. For a given computation $\mathcal{G}$, the computation $\mathcal{G}'$ is the strictification of $\mathcal{G}$ if $\mathcal{G}'$ is a strict computation differing from $\mathcal{G}$ only in its dependency edges and for every dependency edge $(v_i, v_j)$ in $\mathcal{G}$, there is a path from $v_i$ to $v_j$ in the dag of $\mathcal{G}'$. This latter condition ensures that $\mathcal{G}'$ is at least as "strong" as $\mathcal{G}$ in the sense that any execution schedule for $\mathcal{G}'$ is also an execution schedule for $\mathcal{G}$. We now show that for any depth-first computation $\mathcal{G}$, its strictification $\mathcal{G}'$ exists.

**Theorem 3.4** *Every depth-first multithreaded computation can be strictified.*

*Proof:* We strictify a depth-first computation by replacing each nonstrict edge with a strict one. Specifically, let $(v_j, v_l)$ denote a nonstrict edge. See Figure 3.4. Let $\Gamma$ be the least-common ancestor thread of the threads containing $v_j$ and $v_l$. Then let $v_i$ denote the instruction of $\Gamma$ that spawns the subcomputation containing $v_j$, and let $v_k$ denote the instruction of $\Gamma$ that spawns the subcomputation containing $v_l$. Note that $v_i$ and $v_k$ must be different since we only allow a single instruction to perform a single spawn and $\Gamma$ is defined as the least-common ancestor of two different threads. To strictify the computation, we replace dependency edge $(v_j, v_l)$ with a new dependency edge $(v_j, v_k)$ and repeat for every such nonstrict edge. Such a replacement strengthens the dag since there is a path of spawn and continue edges from $v_k$ to $v_l$.

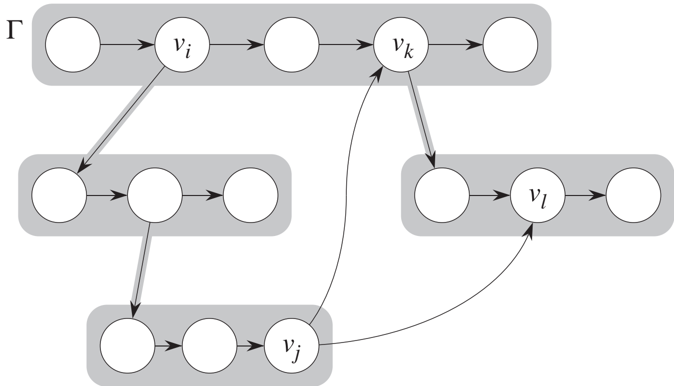



**Figure 3.4**: Every depth-first computation can be strictified by replacing every nonstrict edge $(v_j, v_l)$ with a strict edge $(v_j, v_k)$.

We must show that this transformation produces a strict computation. After the transformation every dependency edge satisfies the strictness condition: every dependency edge goes from a thread to one of its ancestors. Thus, we only need to show that each replacement does not introduce any cycles in the dag. In fact, we show the stronger property that each replacement preserves the depth-first property. Recall that the left-to-right depth-first order, $v_1, v_2, \ldots, v_n$, is defined only in terms of the continue and spawn edges, so it does not change when we replace a dependency edge. When we replace dependency edge $(v_j, v_l)$ with $(v_j, v_k)$, we must show that $j < l$ implies $j < k$. With $j < l$, we must have $v_k$ (the instruction that spawns the subcomputation containing $v_l$) to the right of $v_i$ (the instruction that spawns the subcomputation containing $v_j$) in $\Gamma$. And thus, we have $j < k$. ■

In some cases, strictification may reduce the parallelism by increasing the critical path length. For example, the strictified computation of Figure 3.1(b) has a critical path length of 12 instructions, whereas our original example (nonstrict) computation has a critical path length of 10 instructions. On the other hand, many computations, such as those derived from divide-and-conquer, game-tree search, and many other programs, are naturally strict and still have very short critical paths. In Chapter 5, when we look at the Cilk language and runtime system, we shall see programs that give rise to strict computations with more than ten-thousand-fold parallelism. For now, we focus on exploiting the beneficial structure of strict computations to compute efficient execution schedules.

## 3.3 The busy-leaves property

Once a thread $\Gamma$ has been spawned in a strict computation, a single processor can complete the execution of the entire subcomputation rooted at $\Gamma$ even if no other progress is made on other parts of the computation. In other words, from the time the thread $\Gamma$ is spawned until the time $\Gamma$ dies, there is always at least one thread from the subcomputation rooted at $\Gamma$ that is ready. In particular, no leaf thread in a strict multithreaded computation can stall. As we shall see, this property allows an execution schedule to keep the leaves "busy." By combining this "busy-leaves" property with the greedy property, we derive execution schedules that simultaneously exhibit linear speedup and linear expansion of space.

In this section, we show that for any number $P$ of processors and any strict multithreaded computation with work $T_1$, critical path length $T_\infty$, and stack depth $S_1$, there exists a $P$-processor execution schedule $\mathcal{X}$ that achieves time $T(\mathcal{X}) \leq T_1/P + T_\infty$ and space $S(\mathcal{X}) \leq S_1 P$ simultaneously. We give a simple online $P$-processor parallel algorithm—the Busy-Leaves Algorithm—to compute such a schedule.

The Busy-Leaves Algorithm operates online in the following sense. Before the $t$th step, the algorithm has computed and executed the first $t-1$ steps of the execution schedule. At the $t$th step, the algorithm uses only information from the portion of the computation revealed so far in the execution to compute and execute the $t$th step of the schedule. In particular, it does not use any information from instructions not yet executed or threads not yet spawned.

The Busy-Leaves Algorithm, maintains all living threads in a single thread pool which is uniformly available to all $P$ processors. When spawns occur, new threads are added to this global pool, and when a processor needs work, it removes a ready thread from the pool. Though we describe the algorithm as a $P$-processor parallel algorithm, we shall not analyze it as such. Specifically, in computing the $t$th step of the schedule, we allow each processor to add threads to the thread pool and delete threads from it. Thus, we ignore the effects of processors contending for access to the pool. In fact, we shall only analyze properties of the schedule itself and ignore the cost incurred by the algorithm in computing the schedule. Scheduling overheads come into play in the next chapter.

The Busy-Leaves Algorithm operates as follows. The algorithm begins with the root thread in the global thread pool and all processors idle. At the beginning of each step, each processor either is idle or has a thread to work on. Those processors that are idle, begin the step by attempting to remove any ready thread from the pool. If there are sufficiently many ready threads in the pool to satisfy all of the idle processors, then every idle processor gets a ready thread to work on. Otherwise, some processors remain idle. Then each processor that has a thread to work on executes the next instruction from that thread. In general, once a processor has a thread, call it $\Gamma_a$, to work on, it executes an instruction from $\Gamma_a$ at each step until the thread either spawns, stalls, or dies, in which case, it performs according to the following rules.

❶ **Spawns:** If the thread $\Gamma_a$ spawns a child $\Gamma_b$, then the processor finishes the current step by returning $\Gamma_a$ to the thread pool. The processor begins the next step working on $\Gamma_b$.

❷ **Stalls:** If the thread $\Gamma_a$ stalls, then the processor finishes the current step by returning $\Gamma_a$ to the thread pool. The processor begins the next step idle.

❸ **Dies:** If the thread $\Gamma_a$ dies, then the processor finishes the current step by checking to see if $\Gamma_a$'s parent thread $\Gamma_b$ currently has any living children. If $\Gamma_b$ has no live children and no other processor is working on $\Gamma_b$, then the processor takes $\Gamma_b$ from the pool and begins the next step working on $\Gamma_b$. Otherwise, the processor begins the next step idle.

Figure 3.5 illustrates these three rules in a 2-processor execution schedule computed by the Busy-Leaves Algorithm on the strict computation of Figure 3.1(b). Rule ❶: At step 2, processor $p_1$ working on thread $\Gamma_1$ executes $v_2$ which spawns the child $\Gamma_2$, so $p_1$ places $\Gamma_1$ back in the pool (to be picked up at the beginning of the next step by the idle $p_2$) and begins the next step working on $\Gamma_2$. Rule ❷: At step 11, processor $p_2$ working on thread $\Gamma_1$ executes $v_{19}$ and $\Gamma_1$ stalls, so $p_2$ returns $\Gamma_1$ to the pool and begins the next step idle (and remains idle since the thread pool contains no ready threads). Rule ❸: At step 5, processor $p_1$ working on $\Gamma_3$ executes $v_5$ and $\Gamma_3$ dies, so $p_1$ retrieves the parent $\Gamma_2$ from the pool and begins the next step working on $\Gamma_2$.

| step | thread pool | | processor activity $p_1$ | | $p_2$ | |
|---|---|---|---|---|---|---|
| 1 | | | $\mathbf{\Gamma}_1$: | $v_1$ | | |
| 2 | | | | $v_2$ | | |
| 3 | | | $\mathbf{\Gamma}_2$: | $v_3$ | $\mathbf{\Gamma}_1$: | $v_{14}$ |
| 4 | $\Gamma_1$ | $\mathbf{\Gamma}_2$ | $\mathbf{\Gamma}_3$: | $v_4$ | $\mathbf{\Gamma}_6$: | $v_{15}$ |
| 5 | $\Gamma_1$ | $\mathbf{\Gamma}_2$ | | $v_5$ | | $v_{16}$ |
| 6 | $\Gamma_1$ | | $\mathbf{\Gamma}_2$: | $v_6$ | | $v_{17}$ |
| 7 | $\Gamma_1$ | $\Gamma_2$ | $\mathbf{\Gamma}_4$: | $v_7$ | | |
| 8 | $\Gamma_1$ | $\Gamma_2$ | | $v_8$ | | |
| 9 | $\Gamma_1$ | | $\mathbf{\Gamma}_2$: | $v_9$ | | |
| 10 | | $\Gamma_2$ | $\mathbf{\Gamma}_5$: | $v_{10}$ | $\mathbf{\Gamma}_1$: | $v_{18}$ |
| 11 | | $\Gamma_2$ | | $v_{11}$ | | $v_{19}$ |
| 12 | $\Gamma_1$ | $\Gamma_2$ | | $v_{12}$ | | |
| 13 | $\Gamma_1$ | | $\mathbf{\Gamma}_2$: | $v_{13}$ | | |
| 14 | | | $\mathbf{\Gamma}_1$: | $v_{20}$ | | |

**Figure 3.5**: A 2-processor execution schedule computed by the Busy-Leaves Algorithm for the strict computation of Figure 3.1(b). This schedule lists the living threads in the global thread pool at each step just after each idle processor has removed a ready thread. It also lists the ready thread being worked on and the instruction executed by each of the 2 processors, $p_1$ and $p_2$, at each step. Living threads in the pool that are ready are listed in bold. The other living threads are stalled.

Besides being greedy, for any strict computation, the schedule computed by the Busy-Leaves Algorithm maintains the *busy-leaves property*: at every time step during the execution, every leaf in the spawn subtree has a processor working on it. In other words, at every time step, every living thread that has no living descendants has a processor working on it. We shall now prove this fact and show that it implies linear expansion of space. We begin by showing that any schedule that maintains the busy-leaves property exhibits linear expansion of space.

The space bound of the following lemma accounts for any stack-like memory. In particular, we allow any instruction to allocate memory for its thread provided that the instruction cannot be executed at a time step when its thread has a living child. We allow any instruction to deallocate memory for its thread. Additionally, we require that all memory allocated for a thread is deallocated before the thread dies. At any given time step during the execution, the amount of memory currently allocated for a given living thread is the net memory allocated for the thread by all instructions that have been executed. The memory for a thread need not be allocated in a single contiguous chunk. Nevertheless, we may continue to think of the memory allocated for a thread as being part of an activation frame, though now the frame may grow and shrink as the thread executes. The memory is still stack-like, because we allow the frame to grow only when the thread cannot have any living children. The stack depth $S_1$ of a computation is the amount of memory used by the 1-processor left-to-right depth-first execution.

**Lemma 3.5** *For any multithreaded computation with stack depth $S_1$, any $P$-processor execution schedule $\mathcal{X}$ that maintains the busy-leaves property uses space bounded by $S(\mathcal{X}) \leq S_1 P$.*

*Proof:* The busy-leaves property implies that at all time steps $t$, the spawn subtree has at most $P$ leaves. We bound the space in use at time step $t$ by assigning every living thread to a leaf thread and then showing that the total space currently allocated for all the threads assigned to a given leaf is at most $S_1$. For each living thread, we assign it to a leaf as follows. If the thread is a leaf then it is assigned to itself. Otherwise, the thread is assigned to the same leaf as its rightmost living child (though actually, we can choose any living child). Thus, the threads assigned to any given leaf are all ancestors of that leaf.

Now, consider any leaf thread $\Gamma$ and any ancestor thread $\Gamma'$ that is assigned to $\Gamma$. Let $v$ denote the instruction of $\Gamma$ executed at this time step $t$ (recall that $\Gamma$ is busy), and let $v'$ denote the instruction of $\Gamma'$ that spawned the subcomputation containing $\Gamma$. We claim that the amount of memory currently allocated for $\Gamma'$ is no more than the amount of memory allocated for $\Gamma'$ at the time step in the 1-processor left-to-right depth-first execution when the processor executes instruction $v$ of thread $\Gamma$. We verify the claim by observing that every instruction in $\Gamma'$ to the left of $v'$ has been executed, and though there may also be some instructions to the right of $v'$ that have been executed, none of these latter instructions may allocate memory, since they must have been executed while $\Gamma'$ had a living child. As for thread $\Gamma$ itself, the amount of memory currently allocated for it is equal to the amount of memory allocated for it at the time step in the 1-processor execution when the processor executes instruction $v$.

Thus, the total memory currently allocated for all threads assigned to a given leaf is at most equal to the amount of memory in use by a 1-processor execution when the processor is executing the given leaf, and this amount of memory is at most $S_1$. With a maximum of $P$ leaf threads, the total memory currently in use is at most $S_1 P$, and this bound holds for every time step. ■

The bound $S_1 P$ for schedules that maintain the busy-leaves property is conservative. By charging $S_1$ space for each busy leaf, we may be overcharging. For some computations, by knowing that the schedule preserves the busy-leaves property, we can appeal directly to the fact that the spawn subtree never has more than $P$ leaves to obtain tight bounds on space usage [11].

We finish this chapter by showing that for strict computations, the Busy-Leaves Algorithm computes a schedule that is both greedy and maintains the busy-leaves property. Thus, we show that every strict computation has execution schedules that are simultaneously efficient with respect to time and space.

**Theorem 3.6** *For any number $P$ of processors and any strict multithreaded computation with work $T_1$, critical path length $T_\infty$, and stack depth $S_1$, the Busy-Leaves Algorithm computes a $P$-processor execution schedule $\mathcal{X}$ whose execution time satisfies $T(\mathcal{X}) \leq T_1/P + T_\infty$ and whose space satisfies $S(\mathcal{X}) \leq S_1 P$.*

*Proof:* The time bound follows directly from the greedy-scheduling theorem (Theorem 2.1, page 19), since the Busy-Leaves Algorithm computes a greedy schedule. The space bound follows from Lemma 3.5 if we can show that the Busy-Leaves Algorithm maintains the busy-leaves property. We prove this fact by induction on the number of steps. At the first step of the algorithm, the spawn subtree contains just the root thread which is a leaf, and some processor is working on it. We must show that all of the algorithm rules preserve the busy-leaves property. When a processor has a thread $\Gamma_a$ to work on, it executes instructions from that thread until it either spawns, stalls, or dies. Rule ❶: If $\Gamma_a$ spawns a child $\Gamma_b$, then $\Gamma_a$ is not a leaf (even if it was before) and $\Gamma_b$ is a leaf. In this case, the processor works on $\Gamma_b$, so the new leaf is busy. Rule ❷: If $\Gamma_a$ stalls, then $\Gamma_a$ cannot be a leaf since in a strict computation, the unresolved dependency must come from a descendant. Rule ❸: If $\Gamma_a$ dies, then its parent thread $\Gamma_b$ may turn into a leaf. In this case, the processor works on $\Gamma_b$ unless some other processor already is, so the new leaf is guaranteed to be busy. ■

We now know that every strict multithreaded computatation has an efficient execution schedule and we know how to find it. But these facts take us only so far. Execution schedules must be computed efficiently online, and though the Busy-Leaves Algorithm does compute efficient execution schedules and does operate online, it surely does not do so efficiently, except possibly in the case of small-scale symmetric multiprocessors. This lack of scalability is a consequence of employing a single centralized thread pool at which all processors must contend for access. In the next chapter we present a distributed online scheduling algorithm, and we prove that it is both efficient and scalable.

# Chapter 4

# Work stealing

To execute a multithreaded computation on a parallel computer efficiently, a scheduler must simultaneously keep the processors busy doing work, maintain memory usage within reasonable limits, and avoid communication to the greatest extent possible. In this chapter, we give the first provably efficient online scheduling algorithm for multithreaded computations with dependencies. This algorithm is based on the technique of random "work stealing," in which processors needing work steal computational threads from other processors chosen at random. We show analytically that for fully strict computations, this algorithm is simultaneously efficient with respect to time, space, and communication. In Section 4.1, we present the randomized work-stealing algorithm, give an important structural lemma, and bound the space used by the algorithm. In Section 4.2, we give the model that we use to analyze access contention in the algorithm, and we give a bound on the delay incurred by random accesses in this model. We use the structural lemma of Section 4.1 and the bound of Section 4.2 in Section 4.3 to prove analytically that for fully strict multithreaded computations, this work-stealing algorithm achieves linear speedup with little communication. The Cilk runtime system (Chapter 5) implements this work-stealing algorithm and empirically demonstrates its efficiency.

## 4.1 A randomized work-stealing algorithm

In this section, we present an online, randomized work-stealing algorithm for scheduling multithreaded computations on a parallel computer. Also, we present an important structural lemma which is used at the end of this section to show that for fully strict computa-

The research reported in this chapter is joint work with Charles Leiserson of MIT's Laboratory for Computer Science and was first published in [14].

tions, this algorithm causes at most a linear expansion of space. This lemma reappears in Section 4.3 to show that for fully strict computations, this algorithm achieves linear speedup and generates existentially optimal amounts of communication.

In the Work-Stealing Algorithm, the centralized thread pool of the Busy-Leaves Algorithm is distributed across the processors. Specifically, each processor maintains a *ready deque* data structure of threads. The ready deque has two ends: a *top* and a *bottom*. Threads can be inserted on the bottom and removed from either end. A processor treats its ready deque like a call stack, pushing and popping from the bottom. Threads that are migrated to other processors are removed from the top.

In general, a processor obtains work by removing the thread at the bottom of its ready deque. It starts working on the thread, call it $\Gamma_a$, and continues executing $\Gamma_a$'s instructions until $\Gamma_a$ spawns, stalls, dies, or enables a stalled thread, in which case, it performs according to the following rules.

❶ **Spawns:** If the thread $\Gamma_a$ spawns a child $\Gamma_b$, then $\Gamma_a$ is placed on the bottom of the ready deque, and the processor commences work on $\Gamma_b$.

❷ **Stalls:** If the thread $\Gamma_a$ stalls, its processor checks the ready deque. If the deque contains any threads, then the processor removes and begins work on the bottommost thread. If the ready deque is empty, however, the processor begins work stealing: it steals the topmost thread from the ready deque of a randomly chosen processor and begins work on it. (This work-stealing strategy is elaborated below.)

❸ **Dies:** If the thread $\Gamma_a$ dies, then the processor follows rule ❷ as in the case of $\Gamma_a$ stalling.

❹ **Enables:** If the thread $\Gamma_a$ enables a stalled thread $\Gamma_b$, the now-ready thread $\Gamma_b$ is placed on the bottom of the ready deque of $\Gamma_a$'s processor.

A thread can simultaneously enable a stalled thread and die, in which case we first perform rule ❹ for enabling and then rule ❸ for dying. Except for rule ❹ for the case when a thread enables a stalled thread, these rules are analogous to the rules of the Busy-Leaves Algorithm, and as we shall see, rule ❹ is needed to ensure that the algorithm maintains important structural properties, including the busy-leaves property.

The Work-Stealing Algorithm begins with all ready deques empty. The root thread of the multithreaded computation is placed in the ready deque of one processor, while the other processors start work stealing.

When a processor begins work stealing, it operates as follows. The processor becomes a *thief* and attempts to steal work from a *victim* processor chosen uniformly at random. The

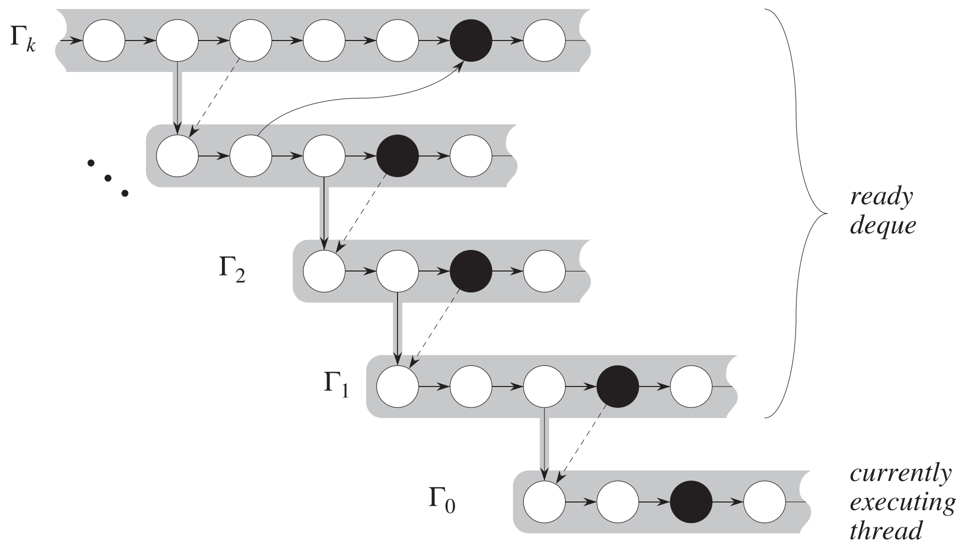



**Figure 4.1**: The structure of a processor's ready deque. The black instruction in each thread indicates the thread's currently ready instruction. Only thread $\Gamma_k$ may have been worked on since it last spawned a child. The dashed edges are the "deque edges" introduced in Section 4.3.

thief queries the ready deque of the victim, and if it is nonempty, the thief removes and begins work on the top thread. If the victim's ready deque is empty, however, the thief tries again, picking another victim at random.

We now state and prove an important lemma on the structure of threads in the ready deque of any processor during the execution of a fully strict computation. This lemma is used later in this section to analyze execution space and in Section 4.3 to analyze execution time and communication. Figure 4.1 illustrates the lemma.

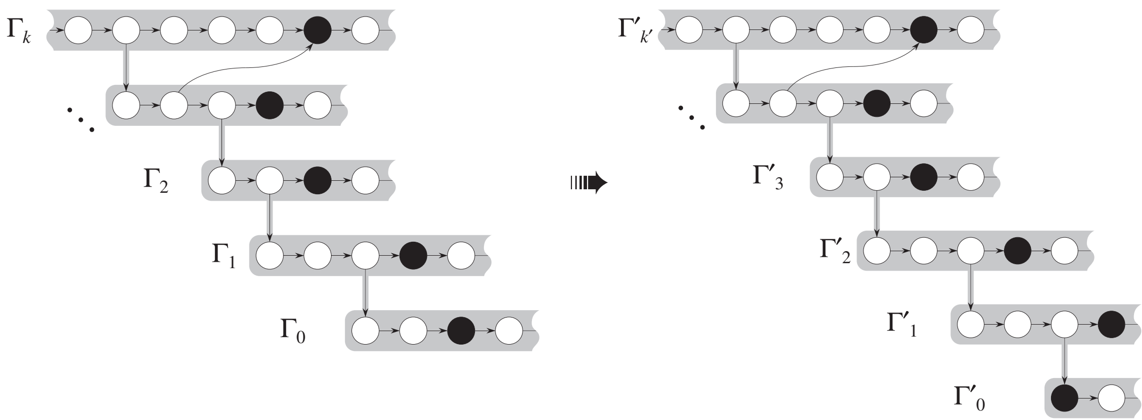
**Lemma 4.1** *In the execution of any fully strict multithreaded computation by the Work-Stealing Algorithm, consider any processor p and any given time step at which p is working on a thread. Let $\Gamma_0$ be the thread that p is working on, let k be the number of threads in p's ready deque, and let $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ denote the threads in p's ready deque ordered from bottom to top, so that $\Gamma_1$ is the bottommost and $\Gamma_k$ is the topmost. If we have $k > 0$, then the threads in p's ready deque satisfy the following properties:*

① *For $i = 1, 2, \ldots, k$, thread $\Gamma_{i-1}$ is a child of $\Gamma_i$.*

② *If we have $k > 1$, then for $i = 1, 2, \ldots, k-1$, thread $\Gamma_i$ has not been worked on since it spawned $\Gamma_{i-1}$.*

*Proof:* The proof is a straightforward induction on execution time. Execution begins with the root thread in some processor's ready deque and all other ready deques empty, so the lemma vacuously holds at the outset. Now, consider any step of the algorithm at which processor $p$ executes an instruction from thread $\Gamma_0$. Let $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ denote the $k$ threads in $p$'s ready deque before the step, and suppose that either $k = 0$ or both properties hold. Let $\Gamma'_0$ denote the thread (if any) being worked on by $p$ after the step, and let $\Gamma'_1, \Gamma'_2, \ldots, \Gamma'_{k'}$ denote the $k'$ threads in $p$'s ready deque after the step. We now look at the rules of the algorithm and show that they all preserve the lemma. That is, either $k' = 0$ or both properties hold after the step.

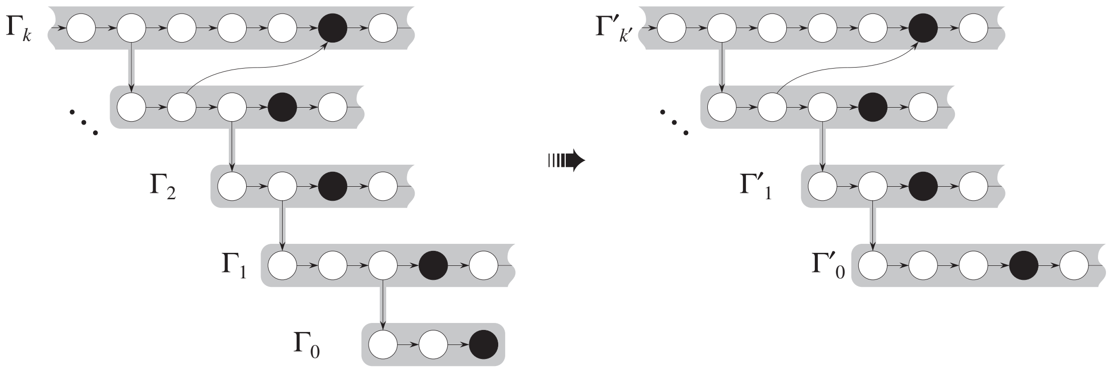
Rule ❶: If $\Gamma_0$ spawns a child, then $p$ pushes $\Gamma_0$ onto the bottom of the ready deque and commences work on the child. Thus, $\Gamma'_0$ is the child, we have $k' = k + 1 > 0$, and for $j = 1, 2, \ldots, k'$, we have $\Gamma'_j = \Gamma_{j-1}$. See Figure 4.2. Now, we can check both properties. Property ①: If $k' > 1$, then for $j = 2, 3, \ldots, k'$, thread $\Gamma'_{j-1}$ is a child of $\Gamma'_j$, since before the spawn we have $k > 0$, which means that for $i = 1, 2, \ldots, k$, thread $\Gamma_{i-1}$ is a child of $\Gamma_i$. Moreover, $\Gamma'_0$ is obviously a child of $\Gamma'_1$. Property ②: If $k' > 2$, then for $j = 2, 3, \ldots, k' - 1$, thread $\Gamma'_j$ has not been worked on since it spawned $\Gamma'_{j-1}$, because before the spawn we have $k > 1$, which means that for $i = 1, 2, \ldots, k - 1$, thread $\Gamma_i$ has not been worked on since it spawned $\Gamma_{i-1}$. Finally, thread $\Gamma'_1$ has not been worked on since it spawned $\Gamma'_0$, because the spawn only just occurred.




**(a)** Before spawn. **(b)** After spawn.

**Figure 4.2**: The ready deque of a processor before and after the thread $\Gamma_0$ that it is working on spawns a child.

Rules ❷ and ❸: If $\Gamma_0$ stalls or dies, then we have two cases to consider. If $k = 0$, the ready deque is empty, so the processor commences work stealing, and when the processor steals and begins work on a thread, we have $k' = 0$. If $k > 0$, the ready deque is not empty, so the processor pops the bottommost thread off the deque and commences work on it. Thus, we have $\Gamma'_0 = \Gamma_1$ (the popped thread) and $k' = k-1$, and for $j = 1,2,\ldots,k'$, we have $\Gamma'_j = \Gamma_{j+1}$. See Figure 4.3. Now, if $k' > 0$, we can check both properties. Property ①: For $j = 1,2,\ldots,k'$, thread $\Gamma'_{j-1}$ is a child of $\Gamma'_j$, since for $i = 1,2,\ldots,k$, thread $\Gamma_{i-1}$ is a child of $\Gamma_i$. Property ②: If $k' > 1$, then for $j = 1,2,\ldots,k'-1$, thread $\Gamma'_j$ has not been worked on since it spawned $\Gamma'_{j-1}$, because before the death we have $k > 2$, which means that for $i = 2,3,\ldots,k-1$, thread $\Gamma_i$ has not been worked on since it spawned $\Gamma_{i-1}$.




**(a)** Before death. **(b)** After death.

**Figure 4.3**: The ready deque of a processor before and after the thread $\Gamma_0$ that it is working on dies.

Rule ❹: If $\Gamma_0$ enables a stalled thread, then due to the fully strict condition, that previously stalled thread must be $\Gamma_0$'s parent. There are two cases to consider. If $k > 0$, then the processor's ready deque is not empty, and this parent thread must be bottommost in the ready deque. In this case, the processor does nothing. If $k = 0$, then the ready deque is empty and the processor places the parent thread on the bottom of the ready deque. In this case, we have $\Gamma'_0 = \Gamma_0$ and $k' = k+1 = 1$ with $\Gamma'_1$ denoting the newly enabled parent. We only have to check the first property. Property ①: Thread $\Gamma'_0$ is obviously a child of $\Gamma'_1$.

If some other processor steals a thread from processor $p$, then we must have $k > 0$, and after the steal we have $k' = k-1$. If $k' > 0$ holds, then both properties are clearly preserved. All other actions by processor $p$—such as work stealing or executing an instruction that does not invoke any of the above rules—clearly preserve the lemma. ∎

Before moving on, it is worth pointing out how it may happen that thread $\Gamma_k$ has been worked on since it spawned $\Gamma_{k-1}$, since Property ② excludes $\Gamma_k$. This situation arises when $\Gamma_k$ is stolen from processor $p$ and then stalls on its new processor. Later, $\Gamma_k$ is reenabled by $\Gamma_{k-1}$ and brought back to processor $p$'s ready deque. The key observation is that when $\Gamma_k$ is reenabled, processor $p$'s ready deque is empty and $p$ is working on $\Gamma_{k-1}$. The other threads $\Gamma_{k-2}, \Gamma_{k-3}, \ldots, \Gamma_0$ shown in Figure 4.1 were spawned after $\Gamma_k$ was reenabled.

We conclude this section by bounding the space used by the Work-Stealing Algorithm executing a fully strict computation. This bound accounts for all stack-like memory.

**Theorem 4.2** *For any fully strict multithreaded computation with stack depth $S_1$, the Work-Stealing Algorithm run on a computer with $P$ processors uses at most $S_1 P$ space.*

*Proof:* Like the Busy-Leaves Algorithm, the Work-Stealing Algorithm maintains the busy-leaves property: at every time step of the execution, every leaf in the current spawn subtree has a processor working on it. If we can establish this fact, then Lemma 3.5 (page 39) completes the proof.

That the Work-Stealing Algorithm maintains the busy-leaves property is a simple consequence of Lemma 4.1. At every time step, every leaf in the current spawn subtree must be ready and therefore must either have a processor working on it or be in the ready deque of some processor. But Lemma 4.1 guarantees that no leaf thread sits in a processor's ready deque while the processor works on some other thread. ∎

With the space bound in hand, we now turn attention to analyzing the time and communication bounds for the Work-Stealing Algorithm. Before we can proceed with this analysis, however, we must take care to define a model for coping with the contention that may arise when multiple thief processors simultaneously attempt to steal from the same victim.

## 4.2 Atomic messages and the recycling game

This section presents the "atomic-access" model that we use to analyze contention during the execution of a multithreaded computation by the Work-Stealing Algorithm. We introduce a combinatorial "balls and bins" game, which we use to bound the total amount of delay incurred by random, asynchronous accesses in this model. We shall use the results of this section in Section 4.3, where we analyze the Work-Stealing Algorithm.

The *atomic-access model* is the machine model we use to analyze the Work-Stealing Algorithm. We assume that the machine is an asynchronous parallel computer with $P$ processors, and its memory can be either distributed or shared. Our analysis assumes that concurrent accesses to the same data structure are serially queued by an adversary, as in the atomic message-passing model of [75]. This assumption is more stringent than that in the model of Karp and Zhang [65]. They assume that if concurrent steal requests are made to a deque, in one time step, one request is satisfied and all the others are denied. In the atomic-access model, we also assume that one request is satisfied, but the others are queued by an adversary, rather than being denied. Moreover, from the collection of waiting requests for a given deque, the adversary gets to choose which is serviced and which continue to wait. The only constraint on the adversary is that if there is at least one request for a deque, then the adversary cannot choose that none be serviced.

The main result of this section is to show that if requests are made randomly by $P$ processors to $P$ deques with each processor allowed at most one outstanding request, then the total amount of time that the processors spend waiting for their requests to be satisfied is likely to be proportional to the total number $M$ of requests, no matter which processors make the requests and no matter how the requests are distributed over time. In order to prove this result, we introduce a "balls and bins" game that models the effects of queueing by the adversary.

The $(P,M)$-*recycling game* is a combinatorial game played by the adversary, in which balls are tossed at random into bins. The parameter $P$ is the number of balls in the game, which is equal to the number of bins. The parameter $M$ is the total number of ball tosses executed by the adversary. Initially, all $P$ balls are in a reservoir separate from the $P$ bins. At each step of the game, the adversary executes the following two operations in sequence:

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

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

The adversary is permitted to make a total of $M$ ball tosses. The game ends when $M$ ball tosses have been made and all balls have been removed from the bins and placed back in the reservoir.

The recycling game models the servicing of steal requests by the Work-Stealing Algorithm. We can view each ball and each bin as being owned by a distinct processor. If a ball is in the reservoir, it means that the ball's owner is not making a steal request. If a ball is in a bin, it means that the ball's owner has made a steal request to the deque of the bin's owner, but that the request has not yet been satisfied. When a ball is removed from a bin and returned to the reservoir, it means that the request has been serviced.

After each step $t$ of the game, there are some number $n_t$ of balls left in the bins, which correspond to steal requests that have not been satisfied. We shall be interested in the *total delay* $D = \sum_{t=1}^{T} n_t$, where $T$ is the total number of steps in the game. The goal of the adversary is to make the total delay as large as possible. The next lemma shows that despite the choices that the adversary makes about which balls to toss into bins and which to return to the reservoir, the total delay is unlikely to be large.

**Lemma 4.3** *For any $\varepsilon > 0$, with probability at least $1-\varepsilon$, the total delay in the $(P,M)$-recycling game is $O(M+P\lg P+P\lg(1/\varepsilon))$.*[1] *The expected total delay is at most $M$. In other words, the total delay incurred by $M$ random requests made by $P$ processors in the atomic-access model is $O(M+P\lg P+P\lg(1/\varepsilon))$ with probability at least $1-\varepsilon$, and the expected total delay is at most $M$.*

*Proof:* We first make the observation that the strategy by which the adversary chooses a ball from each bin is immaterial, and thus, we can assume that balls are queued in their bins in a first-in-first-out (FIFO) order. The adversary removes balls from the front of the queue, and when the adversary tosses a ball, it is placed on the back of the queue. If several balls are tossed into the same bin at the same step, they can be placed on the back of the queue in any order. The reason that assuming a FIFO discipline for queuing balls in a bin does not affect the total delay is that the number of balls in a given bin at a given step is the same no matter which ball is removed, and where balls are tossed has nothing to do with which ball is tossed.

For any given ball and any given step, the step either finishes with the the ball in a bin or in the reservoir. Define the *delay* of ball $r$ to be the random variable $\delta_r$ denoting the total number of steps that finish with ball $r$ in a bin. Then, we have

$$D = \sum_{r=1}^{P} \delta_r \; . \tag{4.1}$$

Define the $i$th *cycle* of a ball to be those steps in which the ball remains in a bin from the $i$th time it is tossed until it is returned to the reservoir. Define also the $i$th *delay* of a ball to be the number of steps in its $i$th cycle.

We shall analyze the total delay by focusing, without loss of generality, on the delay $\delta = \delta_1$ of ball 1. If we let $m$ denote the number of times that ball 1 is tossed by the adversary, and for $i = 1,2,\ldots,m$, let $d_i$ be the random variable denoting the $i$th delay of ball 1, then we have $\delta = \sum_{i=1}^{m} d_i$.

We say that the $i$th cycle of ball 1 is *delayed* by another ball $r$ if the $i$th toss of ball 1 places it in some bin $k$ and ball $r$ is removed from bin $k$ during the $i$th cycle of ball 1. Since the adversary follows the FIFO rule, it follows that the $i$th cycle of ball 1 can be delayed

[1] Greg Plaxton of the University of Texas, Austin has improved this bound to $O(M)$ for the case when $1/\varepsilon$ is at most polynomial in $M$ and $P$ [85].

by another ball $r$ either once or not at all. Consequently, we can decompose each random variable $d_i$ into a sum $d_i = x_{i2} + x_{i3} + \cdots + x_{im}$ of indicator random variables, where

$$x_{ir} = \begin{cases} 1 & \text{if the } i\text{th cycle of ball 1 is delayed by ball } r; \\ 0 & \text{otherwise.} \end{cases}$$

Thus, we have

$$\delta = \sum_{i=1}^{m} \sum_{r=2}^{P} x_{ir} \; . \tag{4.2}$$

We now prove an important property of these indicator random variables. Consider any set $S$ of pairs $(i,r)$, each of which corresponds to the event that the $i$th cycle of ball 1 is delayed by ball $r$. For any such set $S$, we claim that

$$\Pr\left\{ \bigwedge_{(i,r)\in S} (x_{ir} = 1) \right\} \le P^{-|S|} \; . \tag{4.3}$$

The crux of proving the claim is to show that

$$\Pr\left\{ x_{ir} = 1 \;\middle|\; \bigwedge_{(i',r')\in S'} (x_{i'r'} = 1) \right\} \le 1/P \; , \tag{4.4}$$

where $S' = S - \{(i,r)\}$, whence the claim (4.3) follows from Bayes's Theorem.

We can derive Inequality (4.4) from a careful analysis of dependencies. Because the adversary follows the FIFO rule, we have that $x_{ir} = 1$ only if, when the adversary executes the $i$th toss of ball 1, it falls into whatever bin contains ball $r$, if any. *A priori*, this event happens with probability either $1/P$ or 0, and hence, with probability at most $1/P$. Conditioning on any collection of events relating which balls delay this or other cycles of ball 1 cannot increase this probability, as we now argue in two cases. In the first case, the indicator random variables $x_{i'r'}$, where $i' \ne i$, tell whether other cycles of ball 1 are delayed. This information tells nothing about where the $i$th toss of ball 1 goes. Therefore, these random variables are independent of $x_{ir}$, and thus, the probability $1/P$ upper bound is not affected. In the second case, the indicator random variables $x_{ir'}$ tell whether the $i$th toss of ball 1 goes to the bin containing ball $r'$, but this information tells us nothing about whether it goes to the bin containing ball $r$, because the indicator random variables tell us nothing to relate where ball $r$ and ball $r'$ are located. Moreover, no "collusion" among the indicator random variables provides any more information, and thus Inequality (4.4) holds.

Equation (4.2) shows that the delay $\delta$ encountered by ball 1 throughout all of its cycles can be expresses as a sum of $m(P-1)$ indicator random variables. In order for $\delta$ to equal or exceed a given value $\Delta$, there must be some set containing $\Delta$ of these indicator random variables, each of which must be 1. For any specific such set, Inequality (4.3) says that the probability is at most $P^{-\Delta}$ that all random variables in the set are 1. Since there are $\binom{m(P-1)}{\Delta} \leq (emP/\Delta)^{\Delta}$ such sets, we have

$$\begin{aligned}
\Pr\{\delta \geq \Delta\} &\leq \left(\frac{emP}{\Delta}\right)^{\Delta} P^{-\Delta} \\
&= \left(\frac{em}{\Delta}\right)^{\Delta} \\
&\leq \varepsilon/P ,
\end{aligned}$$

whenever $\Delta \geq \max\{2em,\ \lg P + \lg(1/\varepsilon)\}$.

Although our analysis was performed for ball 1, it applies to any other ball as well. Consequently, for any given ball $r$ which is tossed $m_r$ times, the probability that its delay $\delta_r$ exceeds $\max\{2em_r, \lg P + \lg(1/\varepsilon)\}$ is at most $\varepsilon/P$. By Boole's inequality and Equation (4.1), it follows that with probability at least $1-\varepsilon$, the total delay $D$ is at most

$$\begin{aligned}
D &= \sum_{r=1}^{P} \max\{2em_r, \lg P + \lg(1/\varepsilon)\} \\
&= \Theta(M + P\lg P + P\lg(1/\varepsilon)) ,
\end{aligned}$$

since $M = \sum_{r=1}^{P} m_r$.

The upper bound $\mathrm{E}[D] \leq M$ can be obtained as follows. Recall that each $\delta_r$ is the sum of $(P-1)m_r$ indicator random variables, each of which has expectation at most $1/P$. Therefore, by linearity of expectation, $\mathrm{E}[\delta_r] \leq m_r$. Using Equation (4.1) and again using linearity of expectation, we obtain $\mathrm{E}[D] \leq M$. ∎

With this bound on the total delay incurred by $M$ random requests now in hand, we turn back to the Work-Stealing Algorithm.

---

## 4.3 Analysis of the work-stealing algorithm

In this section, we analyze the time and communication cost of executing a fully strict multithreaded computation with the Work-Stealing Algorithm. For any fully strict computation with work $T_1$ and critical path length $T_\infty$, we show that the expected running time with $P$ processors, including scheduling overhead, is $O(T_1/P + T_\infty)$. Moreover, for any $\varepsilon > 0$, the execution time on $P$ processors is $O(T_1/P + T_\infty + \lg P + \lg(1/\varepsilon))$, with probability at least $1-\varepsilon$. We also show that the expected total communication during the execution of a fully strict computation is $O(PT_\infty(1+n_d)S_{\max})$, where $n_d$ is the maximum number of dependency edges from a thread to its parent and $S_{\max}$ is the largest size of any activation frame.

Unlike in the Busy-Leaves Algorithm, the "ready pool" in the Work-Stealing Algorithm is distributed, and so there is no contention at a centralized data structure. Nevertheless, it is still possible for contention to arise when several thieves happen to descend on the same victim simultaneously. In this case, as we have indicated in the previous section, we make the conservative assumption that an adversary serially queues the work-stealing requests.

To analyze the running time of the Work-Stealing Algorithm executing a fully strict multithreaded computation with work $T_1$ and critical path length $T_\infty$ on a computer with $P$ processors, we use an accounting argument. At each step of the algorithm, we collect $P$ dollars, one from each processor. At each step, each processor places its dollar in one of three buckets according to its actions at that step. If the processor executes an instruction at the step, then it places its dollar into the WORK bucket. If the processor initiates a steal attempt at the step, then it places its dollar into the STEAL bucket. And, if the processor merely waits for a queued steal request at the step, then it places its dollar into the WAIT bucket. We shall derive the running time bound by bounding the number of dollars in each bucket at the end of the execution, summing these three bounds, and then dividing by $P$.

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

**Lemma 4.4** *The execution of a fully strict multithreaded computation with work $T_1$ by the Work-Stealing Algorithm on a computer with $P$ processors terminates with exactly $T_1$ dollars in the* WORK *bucket.*

*Proof:* A processor places a dollar in the WORK bucket only when it executes an instruction. Thus, since there are $T_1$ instructions in the computation, the execution ends with exactly $T_1$ dollars in the WORK bucket.

■

Bounding the total dollars in the STEAL bucket requires a significantly more involved "delay-sequence" argument. We first introduce the notion of a "round" of work-steal attempts, and we must also define an augmented dag that we then use to define "critical" instructions. The idea is as follows. If, during the course of the execution, a large number of steals are attempted, then we can identify a sequence of instructions—the delay sequence—in the augmented dag such that each of these steal attempts was initiated while some instruction from the sequence was critical. We then show that a critical instruction is unlikely to remain critical across a modest number of steal attempts. We can then conclude that such a delay sequence is unlikely to occur, and therefore, an execution is unlikely to suffer a large number of steal attempts.

A *round* of work-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 $t$ are also in the same round. We can partition all the steal attempts that occur during an execution into rounds as follows. The first round contains all steal attempts initiated at time steps $1, 2, \ldots, t_1$, where $t_1$ is the earliest time such that at least $3P$ steal attempts were initiated at or before $t_1$. We say that the first round starts at time step 1 and ends at time step $t_1$. In general, if the $i$th round ends at time step $t_i$, then the $(i+1)$st round begins at time step $t_i + 1$ and ends at the earliest time step $t_{i+1} > t_i + 1$ such that at least $3P$ steal attempts were initiated at time steps between $t_i + 1$ and $t_{i+1}$, inclusive. These steal attempts belong to round $i+1$. By definition, each round contains at least $3P$ consecutive steal attempts, and since at most $P-1$ steal attempts can be initiated in a single time step, each round contains fewer than $4P-1$ steal attempts.

The sequence of instructions that make up the delay sequence is defined with respect to an augmented dag obtained by slightly modifying the original dag. Let $G$ denote the original dag, that is, the dag consisting of the computation's instructions as vertices and its continue, spawn, and dependency edges as edges. The augmented dag $G'$ is the original dag $G$ together with some new edges, as follows. For every set of instructions $u$, $v$, and $w$ such that $(u,v)$ is a spawn edge and $(u,w)$ is a continue edge, the *deque* edge $(w,v)$ is placed in $G'$. These deque edges are shown dashed in Figure 4.1. We make the technical assumption that instruction $w$ has no incoming dependency edges, and so $G'$ is a dag. (If a cycle is created, a new instruction between $u$ and $w$ can be created, which does not affect our asymptotic bounds.) If $T_\infty$ is the length of a longest path in $G$, then the longest path in $G'$ has length at most $2T_\infty$. It is worth pointing out that $G'$ is only an analytical tool. The deque edges have no effect on the scheduling and execution of the computation by the Work-Stealing Algorithm.

The deque edges are the key to defining critical instructions. At any time step during the execution, we say that an instruction $v$ is *critical* if every instruction that precedes $v$ (either directly or indirectly) in $G'$ has been executed, that is, if for every instruction $w$ such that there is a directed path from $w$ to $v$ in $G'$, instruction $w$ has been executed. A critical instruction must be ready, since $G'$ contains every edge of $G$, but a ready instruction may or may not be critical. Intuitively, the structural properties of a ready deque enumerated in Lemma 4.1 guarantee that if a thread is deep in a ready deque, then it cannot be critical, because the predecessor of the thread's current instruction across the deque edge has not yet been executed.

We say that a given round of steal attempts *occurs* while instruction $v$ is critical if each of the steal attempts that comprise the round is initiated at a time step when $v$ is critical but is not executed.

We now formalize our definition of a delay sequence.

**Definition 4.5** *A* delay sequence *is a 3-tuple* $(U, R, \Pi)$ *satisfying the following conditions:*

- $U = (u_1, u_2, \ldots, u_L)$ *is a maximal directed path in* $G'$*. Specifically, for* $i = 1, 2, \ldots, L-1$*, the edge* $(u_i, u_{i+1})$ *belongs to* $G'$*, instruction* $u_1$ *has no incoming edges in* $G'$ *(instruction* $u_1$ *must be the first instruction of the root thread), and instruction* $u_L$ *has no outgoing edges in* $G'$ *(instruction* $u_L$ *must be the last instruction of the root thread).*

- $R$ *is a positive integer.*

- $\Pi = (\pi_1, \pi_2, \ldots, \pi_L)$ *is a partition of the integer* $R$*.*

*The delay sequence* $(U, R, \Pi)$ *is said to* occur *during an execution if for each* $i = 1, 2, \ldots, L$*, at least* $\pi_i$ *steal-attempt rounds occur while instruction* $u_i$ *is critical.*

The following lemma states that if a large number of steal attempts take place during an execution, then a delay sequence with large $R$ must occur.

**Lemma 4.6** *Consider the execution of a fully strict multithreaded computation with critical path length* $T_\infty$ *by the Work-Stealing Algorithm on a computer with* $P$ *processors. If at least* $4P(2T_\infty + R)$ *steal attempts occur during the execution, then some* $(U, R, \Pi)$ *delay sequence must occur.*

*Proof:* For a given execution in which at least $4P(2T_\infty + R)$ steal attempts take place, we construct a delay sequence $(U, R, \Pi)$ and show that it occurs. With at least $4P(2T_\infty + R)$ steal attempts, there must be at least $2T_\infty + R$ rounds of steal attempts. We construct the delay sequence by identifying a set of instructions on a directed path in $G'$ such that for every time step during the execution, one of these instructions is critical. There are at most $2T_\infty$ instructions on the delay sequence, so at most $2T_\infty$ steal-attempt rounds could overlap a time step at which one of these instructions gets executed. Therefore, there must be at least $R$ steal-attempt rounds that occur while an instruction from the delay sequence is critical. To finish the proof, we need only produce the directed path $U = (u_1, u_2, \ldots, u_L)$ such that for every time step during the execution, one of the $u_i$ is critical. The partition $\Pi = (\pi_1, \pi_2, \ldots, \pi_L)$ can be derived by simply letting $\pi_i$ equal the number of steal-attempt rounds that occur while $u_i$ is critical.

We work backwards from the last instruction of the root thread, which we denote by $v_1$. Let $w_1$ denote the (not necessarily immediate) predecessor instruction of $v_1$ in $G'$ with the latest execution time. Let $(v_{l_1}, \ldots, v_2, v_1)$ denote a directed path from $w_1 = v_{l_1}$ to $v_1$ in $G'$. We extend this path back to the first instruction of the root thread by iterating this construction as follows. At the $i$th iteration we have an instruction $v_{l_i}$ and a directed path in $G'$ from $v_{l_i}$ to $v_1$. We let $w_{i+1}$ denote the predecessor of $v_{l_i}$ in $G'$ with the latest execution time, and let $(v_{l_{i+1}}, \ldots, v_{l_i+1}, v_{l_i})$, where $v_{l_{i+1}} = w_{i+1}$, denote a directed path from $w_{i+1}$ to $v_{l_i}$ in $G'$. We finish iterating the construction when we get to an iteration $k$ in which $v_{l_k}$ is the first instruction of the root thread. Our desired sequence is then $U = (u_1, u_2, \ldots, u_L)$, where $L = l_k$ and $u_i = v_{L-i+1}$ for $i = 1, 2, \ldots, L$. One can verify that at every time step of the execution, one of the $v_{l_i}$ is critical, and therefore, the sequence has the desired property. ∎

We now establish that a critical instruction is unlikely to remain critical across a modest number of steal-attempt rounds. Specifically, we first show that a critical instruction must be the ready instruction of a thread that is one of the top 2 in its processor's ready deque. We then use this fact to show that after $O(1)$ steal-attempt rounds, a critical instruction is very likely to be executed.

**Lemma 4.7** *At every time step during the execution of a fully strict multithreaded computation by the Work-Stealing Algorithm, each critical instruction must be the ready instruction of a thread that is one of the top 2 in its processor's ready deque.*

*Proof:* Consider any time step, and let $u_0$ be the critical instruction of a thread $\Gamma_0$. Since $u_0$ is critical, $\Gamma_0$ must be ready, and therefore, $\Gamma_0$ must be in the ready deque of some processor $p$. If $\Gamma_0$ is not one of the top 2 threads in $p$'s ready deque, then Lemma 4.1 guarantees that each of the at least 2 threads above $\Gamma_0$ in $p$'s ready deque is an ancestor of $\Gamma_0$. Let $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ denote $\Gamma_0$'s ancestor threads, where $\Gamma_1$ is the parent of $\Gamma_0$ and $\Gamma_k$ is the root thread. Further, for $i = 1, 2, \ldots, k$, let $u_i$ denote the instruction of thread $\Gamma_i$ that spawned thread $\Gamma_{i-1}$, and let $w_i$ denote $u_i$'s successor instruction in thread $\Gamma_i$. In the dag $G'$, we have deque edges $(w_i, u_{i-1})$ for $i = 1, 2, \ldots, k$. Consequently, since $u_0$ is critical, for $i = 1, 2, \ldots, k$, each instruction $w_i$ must have been executed, since it is a predecessor of $u_0$ in $G'$. Moreover, because each $w_i$ is the successor of the spawn instruction $u_i$ in thread $\Gamma_i$, each thread $\Gamma_i$ for $i = 1, 2, \ldots, k$ must have been worked on since the time step at which it spawned thread $\Gamma_{i-1}$. But Lemma 4.1 guarantees that only the topmost thread in $p$'s ready deque can have this property. Thus, $\Gamma_1$ is the only thread that can possibly be above $\Gamma_0$ in $p$'s ready deque. ∎

**Lemma 4.8** *Consider the execution of any fully strict multithreaded computation by the Work-Stealing Algorithm on a parallel computer with $P \geq 2$ processors. For any instruction $v$ and any number $r \geq 4$ of steal-attempt rounds, the probability that $r$ rounds occur while the instruction is critical is at most the probability that only 0 or 1 of the steal attempts initiated in the $r$ rounds choose $v$'s processor, which is at most $e^{-2r}$.*

*Proof:* Let $t_a$ denote the first time step at which instruction $v$ is critical, and let $p$ denote the processor in whose ready deque $v$'s thread resides at time step $t_a$. Suppose $r$ steal-attempt rounds occur while instruction $v$ is critical, and consider the steal attempts that comprise these rounds, of which there must be at least $3rP$. Let $t_b$ denote the time step at which the last of these steal attempts is initiated, which must occur before the time step at which instruction $v$ is executed. At least $3rP - P = (3r-1)P$ of these steal attempts must be initiated at a time step before $t_b$, since fewer than $P$ steal attempts can be initiated at time step $t_b$.

We shall first show that of these $(3r-1)P$ steal attempts initiated while instruction $v$ is critical and at least 2 time steps before $v$ is executed, at most 1 of them can choose processor $p$ as its target, for otherwise, $v$ would be executed at or before $t_b$. Recall from Lemma 4.7 that instruction $v$ is the ready instruction of a thread $\Gamma$, which must be among the top 2 threads in $p$'s ready deque as long as $v$ is critical.

If $\Gamma$ is topmost, then another thread cannot become topmost until after instruction $v$ is executed, since only by processor $p$ executing instructions from $\Gamma$ can another thread be placed on the top of its ready deque. Consequently, if a steal attempt targeting processor $p$ is initiated at some time step $t \geq t_a$, we are guaranteed that instruction $v$ is executed at a time step no later than $t$, either by thread $\Gamma$ being stolen and executed or by $p$ executing the thread itself.

Now, suppose $\Gamma$ is second from the top in $p$'s ready deque with thread $\Gamma'$ on top. In this case, if a steal attempt targeting processor $p$ is initiated at time step $t \geq t_a$, then thread $\Gamma'$ gets stolen from $p$'s ready deque no later than time step $t$. Suppose further that another steal attempt targeting processor $p$ is initiated at time step $t'$, where $t_a \leq t \leq t' < t_b$. Then, we know that a second steal will be serviced by $p$ at or before time step $t'+1$. If this second steal gets thread $\Gamma$, then instruction $v$ must get executed at or before time step $t'+1 \leq t_b$, which is impossible, since $v$ is executed after time step $t_b$. Consequently, this second steal must get thread $\Gamma'$—the same thread that the first steal got. But this scenario can only occur if in the intervening time period, thread $\Gamma'$ stalls and is subsequently reenabled by the execution of some instruction from thread $\Gamma$, in which case instruction $v$ must be executed before time step $t'+1 \leq t_b$, which is once again impossible.

Thus, we must have $(3r-1)P$ steal attempts, each initiated at a time step $t$ such that $t_a \leq t < t_b$, and at most 1 of which targets processor $p$. The probability that either 0 or 1 of

$(3r-1)P$ steal attempts chooses processor $p$ is

$$\begin{aligned}
\left(1-\frac{1}{P}\right)^{(3r-1)P} &+ (3r-1)P\left(\frac{1}{P}\right)\left(1-\frac{1}{P}\right)^{(3r-1)P-1} \\
&\le \left(1-\frac{1}{P}+3r-1\right)\left(1-\frac{1}{P}\right)^{(3r-1)P-1} \\
&\le 3re^{-3r+3/2} \\
&\le e^{-2r}
\end{aligned}$$

for $r \geq 4$. ■

We now complete the delay-sequence argument and bound the total dollars in the STEAL bucket.

**Lemma 4.9** *Consider the execution of any fully strict multithreaded computation with critical path length $T_\infty$ by the Work-Stealing Algorithm on a parallel computer with P processors. For any $\varepsilon > 0$, with probability at least $1-\varepsilon$, at most $O(P(T_\infty + \lg(1/\varepsilon)))$ work-steal attempts occur. The expected number of steal attempts is $O(PT_\infty)$. In other words, with probability at least $1-\varepsilon$, the execution terminates with at most $O(P(T_\infty + \lg(1/\varepsilon)))$ dollars in the* STEAL *bucket, and the expected number of dollars in this bucket is $O(PT_\infty)$.*

*Proof:* From Lemma 4.6, we know that if at least $4P(2T_\infty + R)$ steal attempts occur, then some delay sequence $(U, R, \Pi)$ must occur. Consider a particular delay sequence $(U, R, \Pi)$ having $U = (u_1, u_2, \ldots, u_L)$ and $\Pi = (\pi_1, \pi_2, \ldots, \pi_L)$ where $\pi_1 + \pi_2 + \cdots + \pi_L = R$ and $L \leq 2T_\infty$. We shall compute the probability that $(U, R, \Pi)$ occurs.

Such a sequence occurs if, for each $i = 1, 2, \ldots, L$, at least $\pi_i$ steal-attempt rounds occur while instruction $u_i$ is critical. From Lemma 4.8, we know that the probability of at least $\pi_i$ rounds occurring while a given instruction $u_i$ is critical is at most the probability that only 0 or 1 steal attempts initiated in the $\pi_i$ rounds choose $v$'s processor, which is at most $e^{-2\pi_i}$ provided $\pi_i \geq 4$. (For those values of $i$ with $\pi_i < 4$, we use 1 as an upper bound on this probability.) Moreover, since the targets of the work-steal attempts in the $\pi_i$ rounds are chosen independently from the targets chosen in other rounds, we can bound the probability of the particular delay sequence $(U, R, \Pi)$ occurring as follows:

$$\begin{aligned}
\Pr\{(U,R,\Pi) \text{ occurs}\} &= \prod_{1\le i\le L} \Pr\{\pi_i \text{ rounds occur while } u_i \text{ is critical}\} \\
&\le \prod_{\substack{1\le i\le L \\ \pi_i\ge 4}} e^{-2\pi_i}
\end{aligned}$$

$$
\begin{aligned}
&= \exp\left[-2 \sum_{\substack{1 \le i \le L \\ \pi_i \ge 4}} \pi_i\right] \\
&= \exp\left[-2\left(\sum_{1 \le i \le L} \pi_i - \sum_{\substack{1 \le i \le L \\ \pi_i < 4}} \pi_i\right)\right] \\
&\le e^{-2(R-3L)} .
\end{aligned}
$$

To bound the probability of some $(U,R,\Pi)$ delay sequence occurring, we need to count the number of such delay sequences and multiply by the probability that a particular such sequence occurs. The directed path $U$ in the modified dag $G'$ starts at the first instruction of the root thread and ends at the last instruction of the root thread. If the original dag has degree $d$, then $G'$ has degree at most $d+1$. Consistent with our unit-time assumption for instructions, we assume that the degree $d$ is a constant. Since the length of a longest path in $G'$ is at most $2T_\infty$, there are at most $(d+1)^{2T_\infty}$ ways of choosing the path $U = (u_1, u_2, \ldots, u_L)$. There are at most $\binom{L+R}{R} \le \binom{2T_\infty+R}{R}$ ways to choose $\Pi$, since $\Pi$ partitions $R$ into $L$ pieces. As we have just shown, a given delay sequence has at most an $e^{-2(R-3L)} \le e^{-2(R-6T_\infty)}$ chance of occurring. Multiplying these three factors together bounds the probability that any $(U,R,\Pi)$ delay sequence occurs by

$$
(d+1)^{2T_\infty} \binom{2T_\infty + R}{R} e^{-2R+12T_\infty} , \tag{4.5}
$$

which is at most $\varepsilon$ for $R = \Theta(T_\infty \lg d + \lg(1/\varepsilon))$. Thus, the probability that at least $4P(2T_\infty + R) = \Theta(P(T_\infty \lg d + \lg(1/\varepsilon))) = \Theta(P(T_\infty + \lg(1/\varepsilon)))$ steal attempts occur is at most $\varepsilon$. The expectation bound follows, because the tail of the distribution decreases exponentially. ■

With bounds on the number of dollars in the WORK and STEAL buckets, we now state the theorem that bounds the total execution time for a fully strict multithreaded computation by the Work-Stealing Algorithm, and we complete the proof by bounding the number of dollars in the WAIT bucket.

**Theorem 4.10** *Consider the execution of any fully strict multithreaded computation with work $T_1$ and critical path length $T_\infty$ by the Work-Stealing Algorithm on a parallel computer with $P$ processors. The expected running time, including scheduling overhead, is $O(T_1/P + T_\infty)$. Moreover, for any $\varepsilon > 0$, with probability at least $1-\varepsilon$, the execution time on $P$ processors is $O(T_1/P + T_\infty + \lg P + \lg(1/\varepsilon))$.*[2]

*Proof:* Lemmas 4.4 and 4.9 bound the dollars in the WORK and STEAL buckets, so we now must bound the dollars in the WAIT bucket. This bound is given by Lemma 4.3 which bounds the total delay—that is, the total dollars in the WAIT bucket—as a function of the number $M$ of steal attempts—that is, the total dollars in the STEAL bucket. This lemma says that for any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the number of dollars in the WAIT bucket is at most a constant times the number of dollars in the STEAL bucket plus $O(P \lg P + P \lg(1/\varepsilon))$, and the expected number of dollars in the WAIT bucket is at most the number in the STEAL bucket.

We now add up the dollars in the three buckets and divide by $P$ to complete this proof. ∎

The next theorem bounds the total amount of communication that a multithreaded computation executed by the Work-Stealing Algorithm performs in a distributed model. The analysis makes the assumption that at most a constant number of bytes need be communicated along a dependency edge to resolve the dependency.

**Theorem 4.11** *Consider the execution of any fully strict multithreaded computation with critical path length $T_\infty$ by the Work-Stealing Algorithm on a parallel computer with $P$ processors. Then, the total number of bytes communicated has expectation $O(PT_\infty(1+n_d)S_{max})$ where $n_d$ is the maximum number of dependency edges from a thread to its parent and $S_{max}$ is the size in bytes of the largest activation frame in the computation. Moreover, for any $\varepsilon > 0$, the probability is at least $1 - \varepsilon$ that the total communication incurred is $O(P(T_\infty + \lg(1/\varepsilon))(1+n_d)S_{max})$.*

*Proof:* We prove the bound for the expectation. The high-probability bound is analogous. By our bucketing argument, the expected number of steal attempts is at most $O(PT_\infty)$. When a thread is stolen, the communication incurred is at most $S_{\max}$. Communication also occurs whenever a dependency edge enters a parent thread from one of its children and the parent has been stolen, but since each dependency edge accounts for at most a constant number of bytes, the communication incurred is at most $O(n_d)$ per steal. Finally, we can have communication when a child thread enables its parent and puts the parent into the child's processor's ready deque. This event can happen at most $n_d$ times for each time the parent is stolen, so the communication incurred is at most $n_d S_{\max}$ per steal. Thus, the expected total communication cost is $O(PT_\infty(1+n_d)S_{\max})$. ∎

The communication bounds in this theorem are existentially tight, in that there exist fully strict computations that require $\Omega(PT_\infty(1+n_d)S_{\max})$ total communication for any execution

[2] With Plaxton's bound [85] for Lemma 4.3, this bound becomes $T_P = O(T_1/P + T_\infty)$, whenever $1/\varepsilon$ is at most polynomial in $M$ and $P$.

schedule that achieves linear speedup. This result follows directly from a theorem of Wu and Kung [107], who showed that divide-and-conquer computations—a special case of fully strict computations with $n_d = 1$—require this much communication.

In the case when we have $n_d = O(1)$ and the algorithm achieves linear expected speedup —that is, when $P = O(T_1/T_\infty)$—the total communication is at most $O(T_1 S_{\max})$. Moreover, if $P \ll T_1/T_\infty$, the total communication is much less than $T_1 S_{\max}$, which confirms the folk wisdom that work-stealing algorithms require much less communication than the possibly $\Theta(T_1 S_{\max})$ communication of work-sharing algorithms.

# Chapter 5

# Parallel programming in Cilk

Writing a high-performance parallel application in Cilk, the programmer can focus on expressing the parallelism in the algorithm, insulated from communication protocols, load balancing, and other runtime scheduling issues. The Cilk language is an explicitly parallel, multithreaded extension of the C language. The Cilk runtime system automatically manages the low-level details involved in executing a Cilk program on a parallel machine, and it does so with a work-stealing scheduler that is efficient in theory as well as in practice. Moreover, Cilk gives the user an algorithmic model of application performance based on the measures of "work" and "critical path length" which can be used to predict the runtime of a Cilk program accurately. Consequently, a Cilk programmer can tune the performance of his or her application by focusing on the work and critical path, unworried about machine specific performance details.

In this chapter, we explain the Cilk language and runtime system, and then we demonstrate the efficiency of Cilk's scheduler both empirically and analytically. The language and runtime system are covered in Section 5.1. In Section 5.2 we use several applications to demonstrate empirically the efficiency of the Cilk runtime system. These applications include protein folding, graphic rendering, backtrack search, and the ⋆Socrates chess program, which won second prize in the 1995 ICCA World Computer Chess Championship. We also show in Section 5.2 how work and critical path length can be used to model accurately the parallel runtime of Cilk applications. Analytically, we prove in Section 5.3 that for "fully strict" (well-structured) programs, Cilk's work-stealing scheduler uses space, time, and communication all within a constant factor of optimal. To date, all of the applications that we have coded are fully strict.

Some of the research reported in this chapter is joint work with members of the Cilk team: Matteo Frigo, Michael Halbherr, Chris Joerg, Bradley Kuszmaul, Charles Leiserson, Rob Miller, Keith Randall, and Yuli Zhou all currently or formerly of MIT's Laboratory for Computer Science. Some of the material in this chapter was previously published in [12].

## 5.1 The Cilk language and runtime system

The Cilk language [10] extends C with primitives to express parallelism, and the Cilk runtime system maps the expressed parallelism into parallel execution. A Cilk program is preprocessed to C using the `cilk2c` translator[1] [76] and then compiled and linked with a runtime library to run on the target platform. Currently supported targets include the Connection Machine CM5 MPP, the Intel Paragon MPP, the Sun SparcStation SMP, the Silicon Graphics Power Challenge SMP, and the Cilk-NOW network of workstations (Chapter 6). In this section, we shall discuss the Cilk language primitives for expressing parallelism as well as the runtime system mechanisms that implement these primitives. Beginning with two simple high-level language primitives, "spawn" and "sync," we shall then dive into the lower-level primitives, based on "continuation-passing threads," that are supported directly by the runtime system. We conclude this section by discussing the work-stealing scheduler employed by the runtime system to execute a Cilk program in parallel.

### 5.1.1 Spawns and syncs

A Cilk program contains one or more "Cilk procedures," and Cilk procedures can "spawn" children for parallel execution. A *Cilk procedure* is the parallel equivalent of a C function, and a *spawn* is the parallel equivalent of a function call. A spawn differs from a call in that when a procedure spawns a child, the parent and child may execute concurrently. The Cilk language provides a mechanism to define Cilk procedures, a primitive that procedures may use to spawn children, and a primitive that procedures may use to synchronize with their spawned children.

To illustrate these Cilk constructs, we shall use the double recursive implementation of the Fibonacci function as a running example. Recall that the Fibonacci function $\mathrm{fib}(n)$ for $n \geq 0$ is defined as

$$\mathrm{fib}(n) = \begin{cases} n & \text{if } n < 2; \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{otherwise.} \end{cases}$$

Figure 5.1 shows how this function is written as a Cilk procedure. This toy example illus-

[1] The `cilk2c` translator was written by Rob Miller formerly of MIT's Laboratory for Computer Science and now of the School of Computer Science at Carnegie Mellon University. Rob's implementation builds on earlier work by Yuli Zhou formerly of MIT's Laboratory for Computer Science and now of AT&T Bell Laboratories.

trates a common pattern occurring in (parallel) divide-and-conquer applications: recursive calls (spawns) solve smaller subcases and then the partial results are merged to produce the final result.

```
cilk int Fib (int n)
{  if (n<2)
      return n;
   else
   {  int x, y;
      x = spawn Fib (n-1);
      y = spawn Fib (n-2);
      sync;
      return (x+y);
   }
}
```

**Figure 5.1**: A Cilk procedure to compute the *n*th Fibonacci number.

The keyword `cilk` identifies a Cilk procedure. The definition of a procedure *P* includes a return type, an argument list, and a body just like a C function:

```
cilk type P (arg-decls ...)  { stmts ...}
```

The actual work in a Cilk procedure is performed by ordinary C code in the body, which is executed serially. To express parallelism, a procedure may spawn children using the `spawn` keyword, and it may contain synchronization points identified by the `sync` statement. Notice that besides the keywords `cilk`, `spawn`, and `sync`, the Cilk procedure `Fib` is identical to its C counterpart (Figure 1.1, page 2).

Cilk programs create parallelism at runtime when a procedure spawns children. A Cilk procedure may spawn a child *P* as follows:

```
[var =] spawn P (args ...)
```

Besides the keyword `spawn`, this construct is identical to a C function call. Semantically, the difference is as follows. When a C function calls a child, the parent immediately suspends executing and waits for the child to return before resuming. A C function call is synchronous. On the other hand, when a Cilk procedure spawns a child, the parent need not immediately wait for its child to return. A spawn is asynchronous. For example, when the `Fib` procedure spawns its first child, the parent may continue on to the second spawn while the child may execute concurrently on some other processor. The programmer specifies neither where nor exactly when the spawned child will execute. The scheduler makes these decisions.

A Cilk procedure cannot use the return values of the children it has spawned until those children return. For example, the `Fib` procedure cannot add the return values `x` and `y` until its spawned children have completed computing and returned those values. Therefore, `Fib` uses the `sync` statement after the spawns and before the addition. The `sync` statement forces the procedure to suspend executing and wait until all of its spawned children return. Once all of its children return, the procedure may resume. Thus, the `sync` statement synchronizes the parent with all of its children.

Cilk program execution begins at the Cilk procedure `cilk_main`. Like the C `main` function, `cilk_main` has the prototype

```
cilk int cilk_main (int argc, char *argv[]);
```

and receives the command-line arguments as its parameters. For example, Figure 5.2 gives the definition of `cilk_main` for the Fibonacci program. The Cilk procedures `cilk_main` and `Fib` together constitute a complete Cilk Fibonacci program.

```
cilk int cilk_main (int argc, char *argv[])
{  int n, result;
   n = atoi (argv[1]);
   result = spawn Fib (n);
   sync;
   printf ("Fib (%d) = %d.\n", n, result);
   return 0;
}
```

**Figure 5.2**: The `cilk_main` procedure for the Fibonacci program.

At runtime, a Cilk program grows a *spawn tree* of procedures that unfolds dynamically as procedures spawn children. The spawn tree is rooted at `cilk_main` and in general connects procedures to the children they spawn. For example, Figure 5.3 shows the spawn tree grown by an execution of the Fibonacci program. Notice that we use the term "procedure" to denote both the static Cilk procedure (identified by the keyword `cilk`) that is part of a Cilk program and the dynamic procedure that (as a consequence of a spawn occurring at runtime) is a node in the spawn tree. The spawn tree is analogous to a conventional call tree, and it is equivalent to the spawn tree described in Chapter 2 except that in Cilk we use the term "procedure" instead of "thread." In Cilk terminology, "thread" has a different meaning.

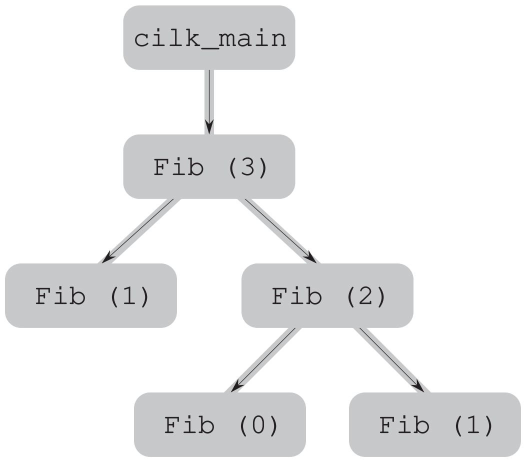



**Figure 5.3**: The spawn tree grown by an execution of the Fibonacci program.

### 5.1.2 Continuation-passing threads

Rather than work with procedures that may suspend waiting to synchronize with their children, the Cilk runtime system manipulates and schedules nonsuspending pieces of procedures called *Cilk threads*. Also, Cilk threads do not interact with each other in spawn/return style as procedures do. Instead, as we shall explain shortly, Cilk threads interact in the "continuation-passing style" supported by the runtime system.

To be executed by the Cilk runtime system, each procedure must be partitioned into one or more threads. The first thread executed as part of a procedure is called the *initial thread*, and subsequent threads are called *successor threads*.

For a Cilk procedure defined with the keyword `cilk`, the `cilk2c` translator automatically partitions the procedure into continuation-passing threads using the `sync` statements as the dividing points. For example, `cilk2c` partitions the `Fib` procedure of Figure 5.1 into two Cilk threads: the initial thread before the `sync` statement and the successor thread after. Currently, `cilk2c` is able to perform this automatic partitioning only for procedures that synchronize exclusively via the `sync` statement which waits for all spawned children.

For procedures that require more complex synchronization, Cilk provides primitives so that the programmer can explicitly define Cilk threads in continuation-passing style. Figure 5.4 shows the Fibonacci function written explicitly as two continuation-passing threads, the initial thread `fib` and its successor `sum`. Essentially, `cilk2c` translates the `Fib` procedure of Figure 5.1 as if it had been written as in Figure 5.4.

The Cilk keyword `thread` identifies a Cilk thread. The definition of a Cilk thread $T$ includes an argument list and a body similar to a C function definition:

```
thread T (arg-decls ...) { stmts ...}
```

```
thread fib (cont int k, int n)
{  if (n<2)
      send_argument (k, n);
   else
   {  cont int x, y;
      spawn_next sum (k, ?x, ?y);
      spawn fib (x, n-1);
      spawn fib (y, n-2);
   }
}

thread sum (cont int k, int x, int y)
{  send_argument (k, x+y);
}
```

**Figure 5.4**: A Cilk procedure, written in explicit continuation-passing style, to compute the *n*th Fibonacci number. This procedure contains two threads, `fib` and its successor `sum`.

The actual work in a Cilk thread is performed by ordinary C code in the body, which is executed serially and without suspending. Threads are not allowed to contain `sync` statements.

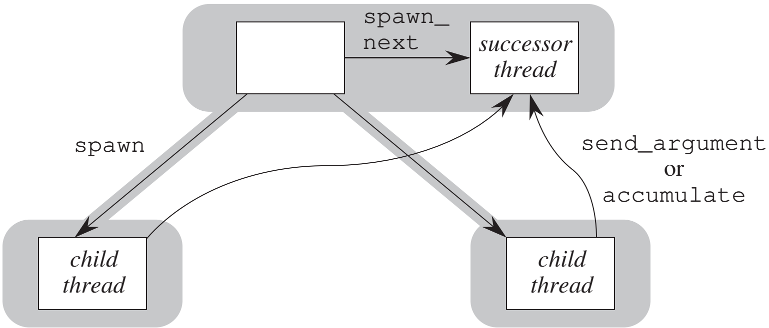



**Figure 5.5**: A thread spawns child threads to create parallelism and a successor thread to wait for the values "returned" by the children. The background shading denotes procedures. The child threads each start a new child procedure, and the successor thread is part of the same procedure as its predecessor.

A Cilk thread generates parallelism at runtime by spawning a child thread that becomes the initial thread of a child procedure. After spawning one or more children, the parent thread cannot then wait for its children to return—in Cilk, threads never suspend. Rather, as illustrated in Figure 5.5, the parent thread must additionally spawn a successor thread to wait for the values "returned" from the children. The spawned successor is part of the same procedure as its predecessor. The child procedures return values to the parent procedure by sending those values to the parent's waiting successor. Thus, a thread may wait to begin executing, but once it begins executing, it cannot suspend. Notice that we use the term "thread" to denote both the static Cilk thread (identified by the keyword `thread`) that is part of a Cilk program and the dynamically spawned thread that occurs at runtime. Spawning successor and child threads is done with the `spawn_next` and `spawn` keywords respectively. Sending a value to a waiting thread is done with the `send_argument` or `accumulate` statement. The Cilk runtime system implements these primitives using two basic data structures, closures and continuations, as illustrated in Figure 5.6.

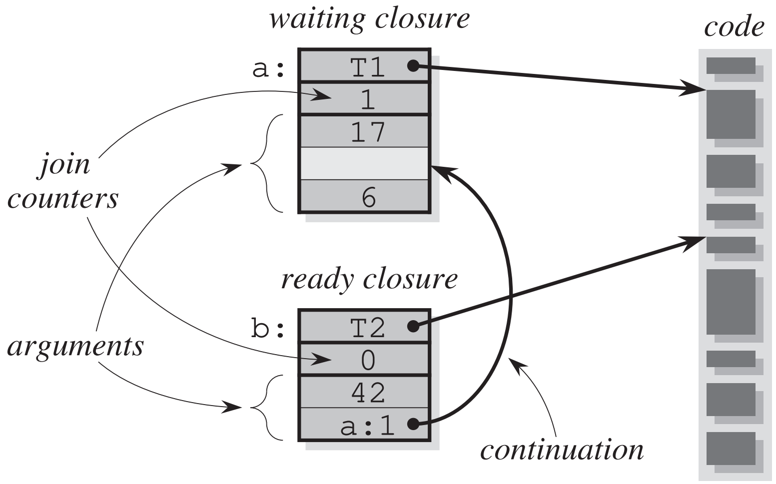



**Figure 5.6**: Closures and continuations.

*Closures* are data structures employed by the runtime system to keep track of and schedule the execution of spawned threads. Whenever a thread is spawned, the runtime system allocates a closure for it from a simple heap. A closure for a thread $T$ consists of a pointer to the code for $T$, a slot for each of $T$'s specified arguments, and a *join counter* indicating the number of missing arguments that need to be supplied before $T$ is ready to run. The closure, or equivalently the spawned thread, is *ready* if it has obtained all of its arguments, and it is *waiting* if some arguments are missing. Figure 5.7 shows a closure for the `fib` thread: its join counter is zero, so the thread is ready, and it contains two arguments, a continuation `k` (as explained below) and an integer `n`. Notice that these two arguments are the formal arguments declared in the definition of the `fib` thread. To run a ready closure, the Cilk scheduler invokes the thread using the values in the closure as arguments. When the thread dies, the closure is freed.

A continuation is a global reference to an empty argument slot of a closure, implemented as a compound data structure containing a pointer to a closure and an offset that designates one of the closure's argument slots (see Figure 5.6). Continuations are typed with the C data type of the slot in the closure. In the Cilk language, continuations are declared by the type modifier keyword `cont`. For example, the `fib` thread declares two integer continuations, `x`

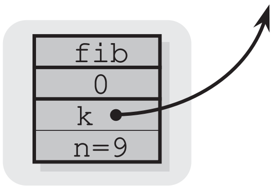



**Figure 5.7**: A closure for the `fib` thread.

and `y`.

A thread spawns a successor thread by creating a closure for the successor. The successor thread is part of the same procedure as its predecessor. Spawning a successor thread $T$ is specified in the Cilk language as follows:

```
spawn_next T (args ...)
```

This statement allocates a closure for thread $T$, fills in all available arguments, and initializes the join counter to the number of missing arguments. Available arguments are specified as in C. To specify a missing argument, the user specifies a continuation variable preceded by a question mark. For example, in the `fib` thread, the statement `spawn_next sum (k, ?x, ?y)` allocates a closure with `sum` as the thread and three argument slots as shown in Figure 5.8. The first slot is initialized with the continuation `k` and the last two slots are empty. The continuation variables `x` and `y` are initialized to refer to these two empty slots, and the join counter is set to 2. This closure is waiting. In general, if the closure is ready, then `spawn_next` causes the closure to be immediately posted to the scheduler for execution. In typical applications, successor threads are spawned waiting for some missing arguments.

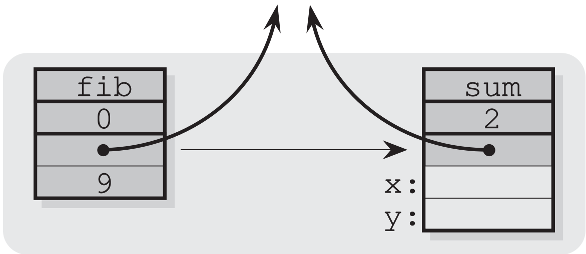



**Figure 5.8**: When the `fib` thread spawns its successor `sum`, it creates a new closure with `sum` as the thread and two empty argument slots referred to by the continuations `x` and `y`. The background shading denotes that both threads are part of the same procedure.

A thread spawns a child thread by creating a closure for the child. The child thread is the initial thread of a newly spawned child procedure. Spawning a child thread $T$ is specified in the Cilk language as follows:

```
spawn T (args ...)
```

This statement is semantically identical to `spawn_next`, but it informs the scheduler that the new closure should be treated as a child, as opposed to a successor. (This difference will be explained when we discuss the scheduler.) For example, the `fib` thread spawns two children as shown in Figure 5.9. The statement `spawn fib (x, n-1)` allocates a closure with `fib` as the thread and two argument slots. The first slot is initialized with the continuation `x` which, as a consequence of the previous statement, refers to a slot in its parent's successor closure. The second slot is initialized with the value of `n-1`. The join counter is set to zero, so the thread is ready, and it is posted to the scheduler for execution. In typical applications, child threads are spawned with no missing arguments.

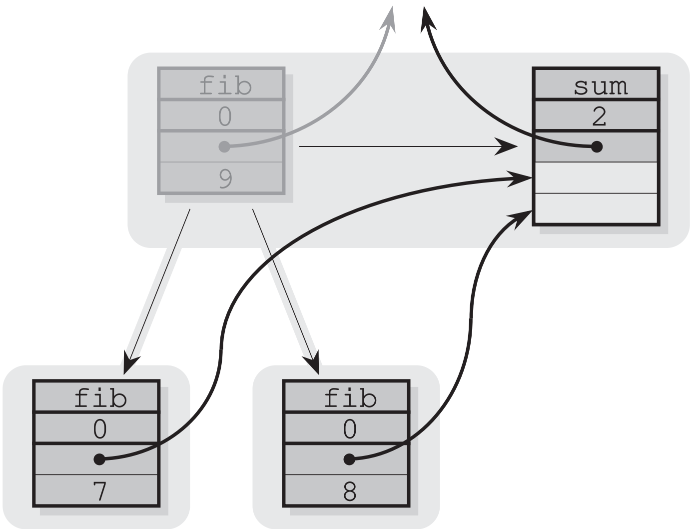



**Figure 5.9**: When the `fib` thread spawns children, it creates for each child a new closure with `fib` as the thread and no empty argument slots. Each child closure has a continuation referring to an empty slot in the parent's successor closure. Each child thread is the initial thread of a new child procedure.

A thread sends a value to a waiting thread by placing the value into an argument slot of the waiting thread's closure. Cilk provides the following primitives to send values from one thread to another:

```
send_argument (k, value)

accumulate (k, op, value)
```

The `send_argument` statement sends the value *value* to the empty argument slot of a waiting closure specified by the continuation *k*. The types of the continuation and the value must be compatible. The join counter of the waiting closure is decremented, and if it becomes zero, then the closure is ready and is posted to the scheduler. For example, when the `fib` thread reaches the boundary case, the statement `send_argument (k, n)` writes the value of `n` into an empty argument slot in the parent procedure's waiting `sum` closure and decrements its join counter. When the `sum` closure's join counter reaches zero, it is posted to the scheduler. When the `sum` thread gets executed, it adds its two arguments, `x` and `y`, and then uses `send_argument` to "return" this result up to its parent procedure's waiting `sum` thread. Cilk also provides a special type of continuation called a `signal` that can be used in a `send_argument` without any *`value`*. The `accumulate` statement is the same as `send_argument` except that it uses the function *`op`* to accumulate *`value`* into the argument slot.

This style of linking Cilk threads is called *continuation-passing style* [2] and contrasts with the spawn/return style that links procedures. In spawn/return style, a spawned child always returns to its parent, and the parent, after performing the spawn, can suspend, waiting for the child to return. That the child should return values and control to its parent is implicit in the spawn/return style. In continuation-passing style, a spawned child never returns to its parent. Cilk threads never `return`, and they never `sync`. Instead, a spawned child is passed a continuation as an explicit argument that specifies where values and control should go when it dies.

At runtime, a Cilk program grows a *dag* (directed, acyclic graph) of threads that unfolds dynamically as threads spawn successors and children. For example, Figure 5.10 shows the dag grown by an execution of the Fibonacci program. The dag contains an edge from one thread to another if either the first thread spawned the second (with `spawn_next` or `spawn`) or the first thread sends a value (or signal) to the second (with `send_argument` or `accumulate`). We can think of closures and continuations as the data structures employed by the runtime system to keep track of the dag as it grows. This dag is analogous to the dag described in Chapter 2 except that rather than having unit-size instructions as dag nodes, this dag has arbitrary-size threads. We shall examine the consequences of this difference in Section 5.3.

To summarize the Fibonacci procedure, it consists of two threads, `fib` and its successor `sum`. Reflecting the explicit continuation passing style, the first argument to each thread is the continuation specifying where the "return value" should be placed. When the `fib` thread is invoked, it first checks to see if the boundary case has been reached, in which case it uses `send_argument` to send the value of `n` as an argument to the waiting thread specified by continuation `k`. Otherwise, it spawns the successor thread `sum`, as well as two children to compute the two subcases. Each of these two children is given a continuation specifying to

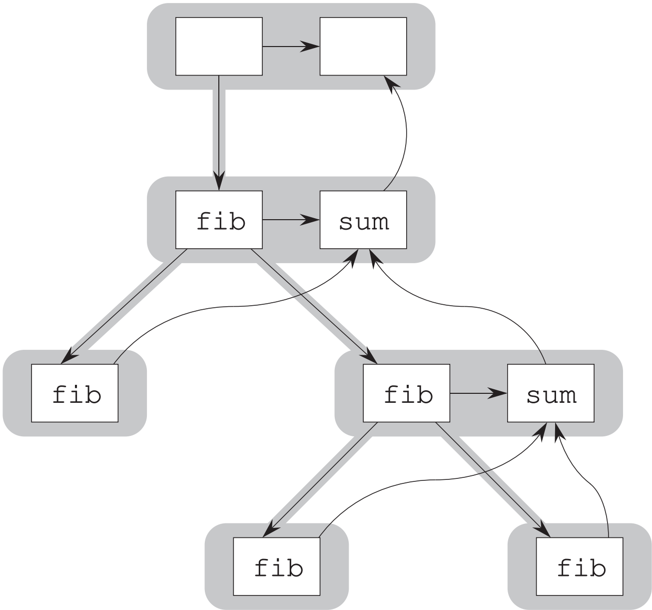



**Figure 5.10**: The dag grown by an execution of the Fibonacci program. Dag edges created by `spawn_next` are horizontal; dag edges created by `spawn` are straight, shaded, and point downward; and dag edges created by `send_argument` are curved and point upwards. The background shading groups the threads into procedures.

which argument in the `sum` thread it should send its result. The `sum` thread simply adds the two arguments when they arrive and sends this result to the waiting thread designated by `k`.

Although writing in explicit continuation passing style is somewhat onerous for the programmer, the decision to break procedures into separate nonsuspending threads with heap-allocated closures simplifies the Cilk runtime system. Each Cilk thread runs to completion without suspending and leaves the C runtime stack empty when it dies. A common alternative [22, 47, 52, 63, 77, 81] is to directly support spawn/return threads (or procedures) in the runtime system, possibly with stack-allocated activation frames. In such a system, threads can suspend waiting for synchronization and leave temporary values on the calling stack. Consequently, this alternative strategy requires that the runtime system either employs multiple stacks or a mechanism to save these temporaries in heap-allocated storage. Another advantage of Cilk's strategy is that it allows multiple children to be spawned from a single nonsuspending thread, which saves on context switching. In Cilk, $r$ children can be spawned and executed with only $r+1$ context switches, whereas the alternative of suspending whenever a thread is spawned causes $2r$ context switches. Since our primary interest is in understanding how to build runtime systems that efficiently schedule multithreaded programs, we chose the alternative of burdening the programmer with a requirement which is perhaps less elegant linguistically, but which yields a simple and portable runtime system implementation.

Cilk supports a variety of additional primitives that give the programmer greater control over runtime performance. For example, when the last action of a thread is to spawn a child, the programmer can use the keyword `call` instead of `spawn` to call the new thread immediately without invoking the scheduler. Additionally, if the called thread is the same as the callee, then the programmer can use the `tail_call` keyword that produces a "tail call" to avoid both the scheduler overhead and the C function call overhead. Cilk also allows arrays and subarrays to be passed as arguments to threads. Other features include various abilities to override the scheduler's decisions, including how to pack and unpack data when a closure is migrated from one processor to another.

### 5.1.3 The Cilk work-stealing scheduler

Cilk's scheduler uses the technique of *work stealing* in which a processor (the thief) who runs out of work selects another processor (the victim) from whom to steal work, and then steals the shallowest ready thread in the victim's spawn tree. Cilk's strategy is for thieves to choose victims at random. Essentially, Cilk implements the scheduling algorithm described and analyzed in Chapter 4. We shall now present Cilk's implementation of this algorithm.

At runtime, each processor maintains a local *ready pool* to hold ready closures. Each closure has an associated *level*, which corresponds to the thread's depth in the spawn tree. The closures for the threads in the `cilk_main` procedure have level 0; the closures for the threads in `cilk_main`'s child procedures have level 1; and so on. The ready pool is an array, illustrated in Figure 5.11, in which the $L$th element contains a linked list of all ready closures having level $L$.

Cilk begins executing the user program by initializing all ready pools to be empty, placing the initial thread of `cilk_main` into the level-0 list of Processor 0's pool, and then starting a scheduling loop on each processor.

At each iteration through the scheduling loop, a processor first checks to see whether its ready pool is empty. If it is, the processor commences work stealing, which will be described shortly. Otherwise, the processor performs the following steps:

1. Remove the closure at the head of the list of the deepest nonempty level in the ready pool.

2. Extract the thread from the closure, and invoke it.

As a thread executes, it may spawn or send arguments to other threads. When the thread dies, control returns to the scheduling loop which advances to the next iteration.

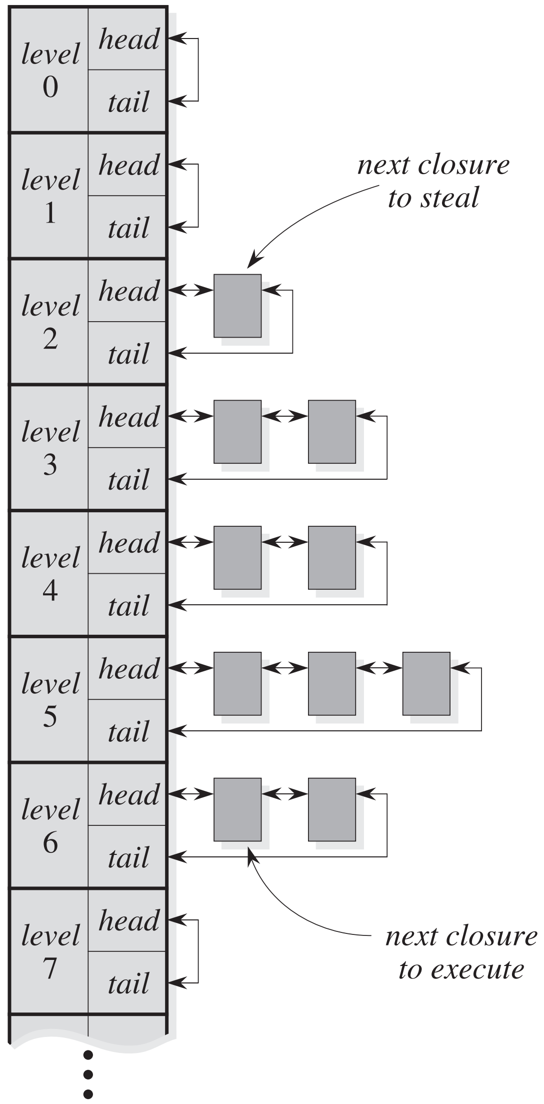



**Figure 5.11**: A processor's ready pool. At each iteration through the scheduling loop, the processor executes the closure at the head of the deepest nonempty level in the ready pool. If the ready pool is empty, the processor becomes a thief and steals the closure at the tail of the shallowest nonempty level in the ready pool of a victim processor chosen uniformly at random.

When a thread at level $L$ performs a `spawn` of a child thread *T*, the processor executes the following operations:

1. Allocate and initialize a closure for *T*.
2. Copy the available arguments into the closure, initialize any continuations to point to missing arguments, and initialize the join counter to the number of missing arguments.
3. Label the closure with level $L+1$.
4. If there are no missing arguments, post the closure to the ready pool by inserting it at the head of the level-$(L+1)$ list.

Execution of `spawn_next` is similar, except that the closure is labeled with level $L$ and, if it is ready, posted to the level-$L$ list.

When a thread performs a `send_argument (`*k*`,` *value*`)`, the processor executes the following operations:

1. Find the closure and argument slot referenced by the continuation *k*.
2. Place *value* in the argument slot, and decrement the join counter of the closure.
3. If the join counter goes to zero, post the closure to the ready pool at the appropriate level.

Execution of `accumulate` is similar, except that the *value* is accumulated into the argument slot with a combining function. When the continuation *k* refers to a closure on a remote processor, network communication ensues. In this case, the processor that initiated the `send_argument` function sends a message to the remote processor to perform the operations. The only subtlety occurs in step 3. If the closure must be posted, it is posted to the ready pool of the initiating processor, rather than to that of the remote processor. This policy is necessary for the scheduler to be provably efficient, but as a practical matter, we have also had success with posting the closure to the remote processor's pool.

If a processor begins an iteration of the scheduling loop and finds that its ready pool is empty, the processor becomes a thief and commences work stealing as follows:

1. Select a victim processor uniformly at random.
2. If the victim's ready pool is empty, go back to step 1.
3. If the victim's ready pool is nonempty, extract the closure from the tail of the list in the shallowest nonempty level of the ready pool, and execute it.

Work stealing is implemented with a simple request-reply communication protocol between the thief and victim.

Why steal work from the shallowest level of the ready pool? The reason is two-fold—one heuristic and one algorithmic. First, to lower communication costs, we would like to steal large amounts of work, and in a tree-structured computation, shallow threads are likely to spawn more work than deep ones. This heuristic notion is the justification cited by earlier researchers [20, 42, 52, 77, 103] who proposed stealing work that is shallow in the spawn tree. We cannot, however, prove that shallow threads are more likely to spawn work than deep ones. What we prove in Section 5.3 is the following algorithmic property. The threads that are on the "critical path" in the dag, are always at the shallowest level of a processor's ready pool. Consequently, if processors are idle, the work they steal makes progress along the critical path.

## 5.2 Performance of Cilk applications

The Cilk runtime system executes Cilk applications efficiently and with predictable performance. Specifically, for dynamic, asynchronous, tree-like applications, Cilk's work-stealing scheduler produces near optimal parallel speedup while using small amounts of space and communication. Furthermore, Cilk application performance can be modeled accurately as a simple function of "work" and "critical path length." In this section, we empirically demonstrate these facts by experimenting with several applications. This section begins with a look at these applications and then proceeds with a look at the performance of these applications. We close this section with a look at application performance modeling. The empirical results of this section confirm the analytical results of the next section.

### 5.2.1 Cilk applications

We experimented with the Cilk runtime system using several applications, some synthetic and some real. The applications are described below:

- `fib` is the same as was presented in Section 5.1, except that the second recursive spawn is replaced by a `tail_call` that avoids the scheduler. This program is a good measure of Cilk overhead, because the thread length is so small.

- `queens` is a backtrack-search program that solves the problem of placing $N$ queens on a $N \times N$ chessboard so that no two queens attack each other. The Cilk program is based on serial code by R. Sargent of MIT's Media Laboratory. Thread length was enhanced by serializing the bottom 7 levels of the search tree.

- `pfold` is a protein-folding program that finds hamiltonian paths in a three-dimensional grid using backtrack search [83]. Written by Chris Joerg of MIT's Laboratory for Computer Science and V. Pande of MIT's Center for Material Sciences and Engineering, `pfold` was the first program to enumerate all hamiltonian paths in a $3 \times 4 \times 4$ grid. We timed the enumeration of all paths starting with a certain sequence.

- `ray` is a parallel program for graphics rendering based on the serial `POV-Ray` program, which uses a ray-tracing algorithm. The entire `POV-Ray` system contains over $20,000$ lines of C code, but the core of `POV-Ray` is a simple doubly nested loop that iterates over each pixel in a two-dimensional image. For `ray` we converted the nested loops into a 4-ary divide-and-conquer control structure using spawns.[2] Our measurements do not include the approximately 2.4 seconds of startup time required to read and process the scene description file.

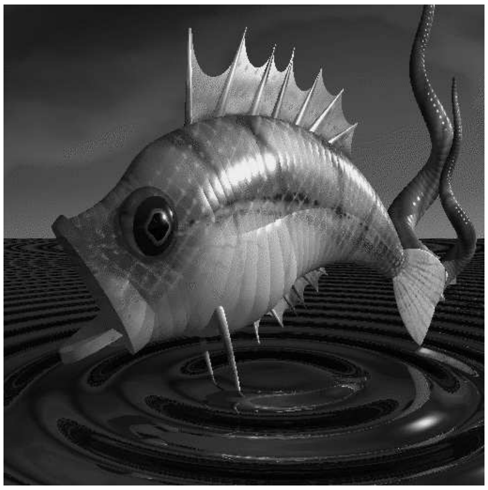
- `knary(n,k,r)` is a synthetic benchmark whose parameters can be set to produce a variety of values for work and critical path length. It generates a tree of depth `n` and branching factor `k` in which the first `r` children at every level are executed serially and the remainder are executed in parallel. At each node of the tree, the program runs an empty "for" loop for 400 iterations.

- `⋆Socrates` is a parallel chess program that uses the Jamboree search algorithm [58, 70] to parallelize a minmax tree search. The work of the algorithm varies with the number of processors, because it does speculative work that may be aborted during runtime. `⋆Socrates` was written by a team of engineers headed by Charles Leiserson of MIT's Laboratory for Computer Science with Don Dailey formerly of Heuristic Software and Chris Joerg of MIT's Laboratory for Computer Science as the lead programmers.[3] `⋆Socrates` won second prize in the 1995 ICCA World Computer Chess Championship running on the 1824-node Intel Paragon at Sandia National Laboratories.

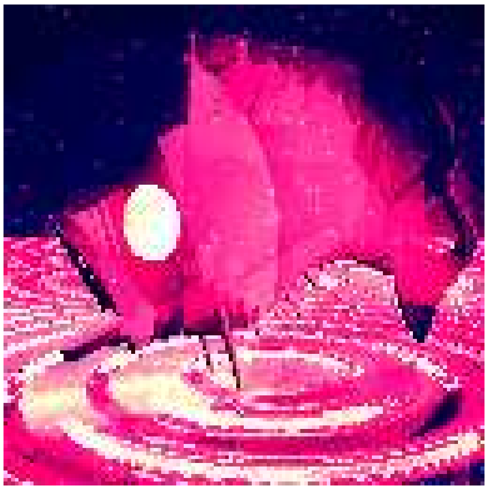
[2] Initially, the Cilk `ray` program was about 5 percent faster than the serial `POV-Ray` program running on one processor. The reason was that the divide-and-conquer decomposition performed by the Cilk code provides better locality than the doubly nested loop of the serial code. Modifying the serial code to imitate the Cilk decomposition improved its performance. Timings for the improved version are given in Figure 5.13.

[3] The other members of the `⋆Socrates` team are I.M. Larry Kaufmann formerly of Heuristic Software, Robert Blumofe of MIT's Laboratory for Computer Science, Bradley Kuszmaul formerly of MIT's Laboratory for Computer Science and now of the Computer Science Department at Yale University, Rolf Riesen of Sandia National Laboratories, and Yuli Zhou formerly of MIT's Laboratory for Computer Science and now of AT&T Bell Laboratories.

Many of these applications place heavy demands on the runtime system due to their dynamic and irregular nature. For example, in the case of queens and pfold, the size and shape of the backtrack-search tree cannot be determined without actually performing the search, and the shape of the tree often turns out to be highly irregular. With speculative work that may be aborted, the ⋆Socrates minmax tree carries this dynamic and irregular structure to the extreme. In the case of ray, the amount of time it takes to compute the color of a pixel in an image is hard to predict and may vary widely from pixel to pixel, as illustrated in Figure 5.12. In all of these cases, high performance demands efficient, dynamic load balancing at runtime.



**(a)** Ray-traced image.



**(b)** Work at each pixel.

**Figure 5.12**: **(a)** An image rendered with the ray program. **(b)** This image shows the amount of time ray took to compute each pixel value. The whiter the pixel, the longer ray worked to compute the corresponding pixel value.

All experiments were run on a CM5 supercomputer. The CM5 is a massively parallel computer based on 32MHz SPARC processors with a fat-tree interconnection network [72]. The Cilk runtime system on the CM5 performs communication among processors using the Strata [17] active-message library.

## 5.2.2 Application performance

By running our applications and measuring a suite of performance parameters, we empirically answer a number of questions about the effectiveness of the Cilk runtime system. We focus on the following questions. How efficiently does the runtime system implement the language primitives? As we add processors, how much faster will the program run? How much more space will it require? And how much more communication will it perform? We show that, for dynamic, asynchronous, tree-like programs, the Cilk runtime system efficiently implements the language primitives, and is simultaneously efficient with respect to time, space, and communication. In Section 5.3, we reach the same conclusion by analytic means, but in this section we focus on empirical data from the execution of our Cilk programs.

The execution of a Cilk program with a given set of inputs grows a *Cilk computation* that consists of a tree of procedures and a dag of threads. These structures were introduced in the previous section. We benchmark our applications based on two fundamental measures of the computation: work and critical path length.

The *work*, denoted by $T_1$, is the time to execute the Cilk computation on one processor, which corresponds to the sum of the execution times of all the threads in the dag. The method used to measure $T_1$ depends on whether the program is deterministic. For deterministic programs, the computation only depends on the program and its inputs, and hence, it is independent of the number of processors and runtime scheduling decisions.[4] All of our applications, except `⋆Socrates`, are deterministic. For these deterministic applications, the work performed by any $P$-processor run of the program is equal to the work performed by a 1-processor run (with the same input values), so we measure the work $T_1$ directly by timing the 1-processor run. The `⋆Socrates` program, on the other hand, uses speculative execution, and therefore, the computation depends on the number of processors and scheduling decisions made at runtime. In this case, timing a 1-processor run is not a reasonable way to measure the work performed by a run with more processors. We must realize that the work $T_1$ of an execution with $P$ processors is defined as the time it takes 1-processor to execute the same *computation*, not the same program (with the same inputs). For `⋆Socrates` we estimate the work of a $P$-processor run by performing the $P$-processor run and timing the execution of every thread and summing. This method yields an underestimate since it does not include scheduling costs. In either case, a $P$-processor execution of a Cilk computation with work $T_1$ must take time at least $T_1/P$.[5] A $P$-processor execution that takes time equal to this $T_1/P$ lower bound is said to achieve *perfect linear speedup*.

The *critical path length*, denoted by $T_\infty$, is the time to execute the Cilk computation with infinitely many processors, which corresponds to the largest sum of thread execution times along any path in the dag. We measure critical path length by timestamping each thread in the dag with the earliest time at which it could have been executed. Specifically this times-

[4]Randomized programs are deterministic if we consider the sequence of values generated by the source of randomness to be inputs to the program.

[5]In practice, we sometimes beat the $T_1/P$ lower bound. Such *superlinear speedup* is a consequence of the fact that as we add processors, we also add other physical resources such as registers, cache, and main memory.

| | `fib` (33) | `queens` (15) | `pfold` (3,3,4) | `ray` (500,500) | `knary` (10,5,2) | `knary` (10,4,1) | ⋆`Socrates` (depth 10) (32 proc.) | ⋆`Socrates` (depth 10) (256 proc) |
|---|---|---|---|---|---|---|---|---|
| (computation parameters) | | | | | | | | |
| $T_{\text{serial}}$ | 8.487 | 252.1 | 615.15 | 729.2 | 288.6 | 40.993 | 1665 | 1665 |
| $T_1$ | 73.16 | 254.6 | 647.8 | 732.5 | 314.6 | 45.43 | 3644 | 7023 |
| $T_{\text{serial}}/T_1$ | 0.116 | 0.9902 | 0.9496 | 0.9955 | 0.9174 | 0.9023 | 0.4569 | 0.2371 |
| $T_\infty$ | 0.000326 | 0.0345 | 0.04354 | 0.0415 | 4.458 | 0.255 | 3.134 | 3.24 |
| $T_1/T_\infty$ | 224417 | 7380 | 14879 | 17650 | 70.56 | 178.2 | 1163 | 2168 |
| threads | 17,108,660 | 210,740 | 9,515,098 | 424,475 | 5,859,374 | 873,812 | 26,151,774 | 51,685,823 |
| thread length | 4.276µs | 1208µs | 68.08µs | 1726µs | 53.69µs | 51.99µs | 139.3µs | 135.9µs |
| (32-processor experiments) | | | | | | | | |
| $T_P$ | 2.298 | 8.012 | 20.26 | 21.68 | 15.13 | 1.633 | 126.1 | - |
| $T_1/P+T_\infty$ | 2.287 | 7.991 | 20.29 | 22.93 | 14.28 | 1.675 | 117.0 | - |
| $T_1/T_P$ | 31.84 | 31.78 | 31.97 | 33.79 | 20.78 | 27.81 | 28.90 | - |
| $T_1/(P\cdot T_P)$ | 0.9951 | 0.9930 | 0.9992 | 1.0558 | 0.6495 | 0.8692 | 0.9030 | - |
| space/proc. | 70 | 95 | 47 | 39 | 41 | 42 | 386 | - |
| requests/proc. | 185.8 | 48.0 | 88.6 | 218.1 | 92639 | 3127 | 23484 | - |
| steals/proc. | 56.63 | 18.47 | 26.06 | 79.25 | 18031 | 1034 | 2395 | - |
| (256-processor experiments) | | | | | | | | |
| $T_P$ | 0.2892 | 1.045 | 2.590 | 2.765 | 8.590 | 0.4636 | - | 34.32 |
| $T_1/P+T_\infty$ | 0.2861 | 1.029 | 2.574 | 2.903 | 5.687 | 0.4325 | - | 30.67 |
| $T_1/T_P$ | 253.0 | 243.7 | 250.1 | 265.0 | 36.62 | 98.00 | - | 204.6 |
| $T_1/(P\cdot T_P)$ | 0.9882 | 0.9519 | 0.9771 | 1.035 | 0.1431 | 0.3828 | - | 0.7993 |
| space/proc. | 66 | 76 | 47 | 32 | 48 | 40 | - | 405 |
| requests/proc. | 73.66 | 80.40 | 97.79 | 82.75 | 151803 | 7527 | - | 30646 |
| steals/proc. | 24.10 | 21.20 | 23.05 | 18.34 | 6378 | 550 | - | 1540 |

**Figure 5.13**: Performance of Cilk on various applications. All times are in seconds, except where noted.

tamp is the maximum of the earliest time that the thread could have been spawned and, for each argument, the earliest time that the argument could have been sent. These values, in turn, are computed from the timestamp of the thread that performed the spawn or sent the argument. In particular, if a thread performs a spawn, then the earliest time that the spawn could occur is equal to the earliest time at which the thread could have been executed (its timestamp) plus the amount of time the thread ran for until it performed the spawn. The same property holds for the earliest time that an argument could be sent. The initial thread of `cilk_main` is timestamped zero, and the critical path length is then computed as the maximum over all threads of its timestamp plus the amount of time it executes for. The measured critical path length does not include scheduling and communication costs. A $P$-processor execution of a Cilk computation must take at least as long as the computation's critical path length $T_\infty$. Thus, if $T_\infty$ exceeds $T_1/P$, then perfect linear speedup cannot be achieved.

Figure 5.13 is a table showing typical performance measures for our Cilk applications. Each column presents data from a single run of a benchmark application. We adopt the following notations. For each application, we have an efficient serial C implementation, compiled using `gcc -O2`, whose measured runtime is denoted $T_{\text{serial}}$. The Cilk computation's work $T_1$ and critical path length $T_\infty$ are measured on the CM5 as described above. The measured execution time of the Cilk program running on $P$ processors of the CM5 is given by $T_P$. The row labeled "threads" indicates the number of threads executed, and "thread length" is the average thread length (work divided by the number of threads).

Certain derived parameters are also displayed in the table. The ratio $T_{\text{serial}}/T_1$ is the *efficiency* of the Cilk program relative to the C program. The ratio $T_1/T_\infty$ is the *average parallelism*. The value $T_1/P + T_\infty$ is a simple model of the runtime, which will be discussed later. The *speedup* is $T_1/T_P$, and the *parallel efficiency* is $T_1/(P \cdot T_P)$. The row labeled "space/proc." indicates the maximum number of closures allocated at any time on any processor. The row labeled "requests/proc." indicates the average number of steal requests made by a processor during the execution, and "steals/proc." gives the average number of closures actually stolen.

The data in Figure 5.13 shows two important relationships: one between efficiency and thread length, and another between speedup and average parallelism.

Considering the relationship between efficiency $T_{\text{serial}}/T_1$ and thread length, we see that for programs with moderately long threads, the Cilk runtime system induces little overhead. The `queens`, `pfold`, `ray`, and `knary` programs have threads with average length greater than 50 microseconds and have efficiency greater than 90 percent. On the other hand, the `fib` program has low efficiency, because the threads are so short: `fib` does almost nothing besides `spawn` and `send_argument`.

Despite it's long threads, the ⋆Socrates program has low efficiency, because its parallel Jamboree search algorithm is based on speculatively searching subtrees that are not searched by a serial algorithm. Consequently, as we increase the number of processors, the program executes more threads and, hence, does more work. For example, the 256-processor execution did 7023 seconds of work whereas the 32-processor execution did only 3644 seconds of work. Both of these executions did considerably more work than the serial program's 1665 seconds of work. Thus, although we observe low efficiency, it is due to the parallel algorithm and not to Cilk overhead.

Looking at the speedup $T_1/T_P$ measured on 32 and 256 processors, we see that when the average parallelism $T_1/T_\infty$ is large compared with the number $P$ of processors, Cilk programs achieve nearly perfect linear speedup, but when the average parallelism is small, the speedup is much less. The `fib`, `queens`, `pfold`, and `ray` programs, for example, have in excess of 7000-fold parallelism and achieve more than 99 percent of perfect linear speedup on 32 processors and more than 95 percent of perfect linear speedup on 256 processors.[6] The ⋆Socrates program exhibits somewhat less parallelism and also somewhat less speedup. On 32 processors the ⋆Socrates program has 1163-fold parallelism, yielding 90 percent of perfect linear speedup, while on 256 processors it has 2168-fold parallelism yielding 80 percent of perfect linear speedup. With even less parallelism, as exhibited in the `knary` benchmarks, less speedup is obtained. For example, the `knary(10,5,2)` benchmark exhibits only 70-fold parallelism, and it realizes barely more than 20-fold speedup on 32 processors (less than 65 percent of perfect linear speedup). With 178-fold parallelism, `knary(10,4,1)` achieves 27-fold speedup on 32 processors (87 percent of perfect linear speedup), but only 98-fold speedup on 256 processors (38 percent of perfect linear speedup).

Although these speedup measures reflect the Cilk scheduler's ability to exploit parallelism, to obtain *application speedup*, we must factor in the efficiency of the Cilk program compared with the serial C program. Specifically, the application speedup $T_{\text{serial}}/T_P$ is the product of efficiency $T_{\text{serial}}/T_1$ and speedup $T_1/T_P$. For example, applications such as `fib` and ⋆Socrates with low efficiency generate correspondingly low application speedup. The ⋆Socrates program, with efficiency 0.2371 and speedup 204.6 on 256 processors, exhibits application speedup of $0.2371 \cdot 204.6 = 48.51$. For the purpose of understanding scheduler performance, we prefer to decouple the efficiency of the application from the efficiency of the scheduler.

Looking more carefully at the cost of a `spawn` in Cilk, we find that it takes a fixed overhead of about 50 cycles to allocate and initialize a closure, plus about 8 cycles for each word

[6] In fact, the `ray` program achieves superlinear speedup even when comparing to the efficient serial implementation. We suspect that cache effects cause this phenomenon.

argument. In comparison, a C function call on a CM5 SPARC processor takes 2 cycles of fixed overhead (assuming no register window overflow) plus 1 cycle for each word argument (assuming all arguments are transferred in registers). Thus, a `spawn` in Cilk is roughly an order of magnitude more expensive than a C function call. This Cilk overhead is quite apparent in the `fib` program, which does almost nothing besides `spawn` and `send_argument`. Based on `fib`'s measured efficiency of 0.116, we can conclude that the aggregate average cost of a `spawn/send_argument` in Cilk is between 8 and 9 times the cost of a function call/return in C.

Efficient execution of programs with short threads requires a low-overhead spawn operation. As can be observed from Figure 5.13, the vast majority of threads execute on the same processor on which they are spawned. For example, the `fib` program executed over 17 million threads but migrated only 6170 (24.10 per processor) when run with 256 processors. Taking advantage of this property, other researchers [47, 63, 77] have developed techniques for implementing spawns such that when the child thread executes on the same processor as its parent, the cost of the spawn operation is roughly equal the cost of a function call. We hope to incorporate such techniques into future implementations of Cilk.

Finally, we make two observations concerning the space and communication measures in Figure 5.13.

Looking at the "space/proc." rows, we observe that the space per processor is generally quite small and does not grow with the number of processors. For example, `⋆Socrates` on 32 processors executes over 26 million threads, yet no processor ever has more than 386 allocated closures. On 256 processors, the number of executed threads nearly doubles to over 51 million, but the space per processor barely changes. In Section 5.3 we show formally that for an important class of Cilk programs, the space per processor does not grow as we add processors.

Looking at the "requests/proc." and "steals/proc." rows in Figure 5.13, we observe that the amount of communication grows with the critical path length but does not grow with the work. For example, `fib`, `queens`, `pfold`, and `ray` all have critical path lengths under a tenth of a second long and perform fewer than 220 requests and 80 steals per processor, whereas `knary(10,5,2)` and `⋆Socrates` have critical path lengths more than 3 seconds long and perform more than 20,000 requests and 1500 steals per processor. The table does not show any clear correlation between work and either requests or steals. For example, `ray` does more than twice as much work as `knary(10,5,2)`, yet it performs two orders of magnitude fewer requests. In Section 5.3, we show that for a class of Cilk programs, the communication per processor grows at most linearly with the critical path length and does not grow as a function of the work.

### 5.2.3 Performance modeling

We further document the effectiveness of the Cilk scheduler by showing empirically that Cilk application performance can be modeled accurately with a simple function of work $T_1$ and critical path length $T_\infty$. Specifically, we use the knary synthetic benchmark to show that the runtime of an application on $P$ processors can be modeled as $T_P \approx T_1/P + c_\infty T_\infty$, where $c_\infty$ is a small constant (about 1.5 for knary) determined by curve fitting. This result shows that we obtain nearly perfect linear speedup when the critical path is short compared with the average amount of work per processor. We also show that a model of this kind is accurate even for ⋆Socrates, which is our most complex application programmed to date.

We would like our scheduler to execute a Cilk computation with $T_1$ work in $T_1/P$ time on $P$ processors. Such perfect linear speedup cannot be obtained whenever the computation's critical path length $T_\infty$ exceeds $T_1/P$, since we always have $T_P \geq T_\infty$ or more generally, $T_P \geq \max\{T_1/P, T_\infty\}$. The critical path length $T_\infty$ is the stronger lower bound on $T_P$ whenever $P$ exceeds the average parallelism $T_1/T_\infty$, and $T_1/P$ is the stronger bound otherwise. A good scheduler should meet each of these bounds as closely as possible.

In order to investigate how well the Cilk scheduler meets these two lower bounds, we used our synthetic knary benchmark, which can grow computations that exhibit a range of values for work and critical path length.

Figure 5.14 shows the outcome from many experiments of running knary with various input values (n, k, and r) on various numbers of processors. The figure plots the measured speedup $T_1/T_P$ for each run against the machine size $P$ for that run. In order to compare the outcomes for runs with different input values, we have normalized the plotted value for each run as follows. In addition to the speedup, we measure for each run the work $T_1$ and the critical path length $T_\infty$, as previously described. We then normalize the machine size and the speedup by dividing these values by the average parallelism $T_1/T_\infty$. For each run, the horizontal position of the plotted datum is $P/(T_1/T_\infty)$, and the vertical position of the plotted datum is $(T_1/T_P)/(T_1/T_\infty) = T_\infty/T_P$. Consequently, on the horizontal axis, the normalized machine-size is 1.0 when the average parallelism is equal to the number of processors. On the vertical axis, the normalized speedup is 1.0 when the runtime equals the critical path length. We can draw the two lower bounds on time as upper bounds on speedup. The horizontal line at 1.0 is the upper bound on speedup obtained from the critical path, $T_P \geq T_\infty$, and the 45-degree line is the linear speedup bound, $T_P \geq T_1/P$. As can be seen from the figure, on the knary runs for which the average parallelism exceeds the number of processors (normalized machine size less than 1), the Cilk scheduler obtains nearly perfect linear speedup. In the region where the number of processors is large compared to the average parallelism (normalized machine size greater than 1), the data is more scattered, but the speedup is al-

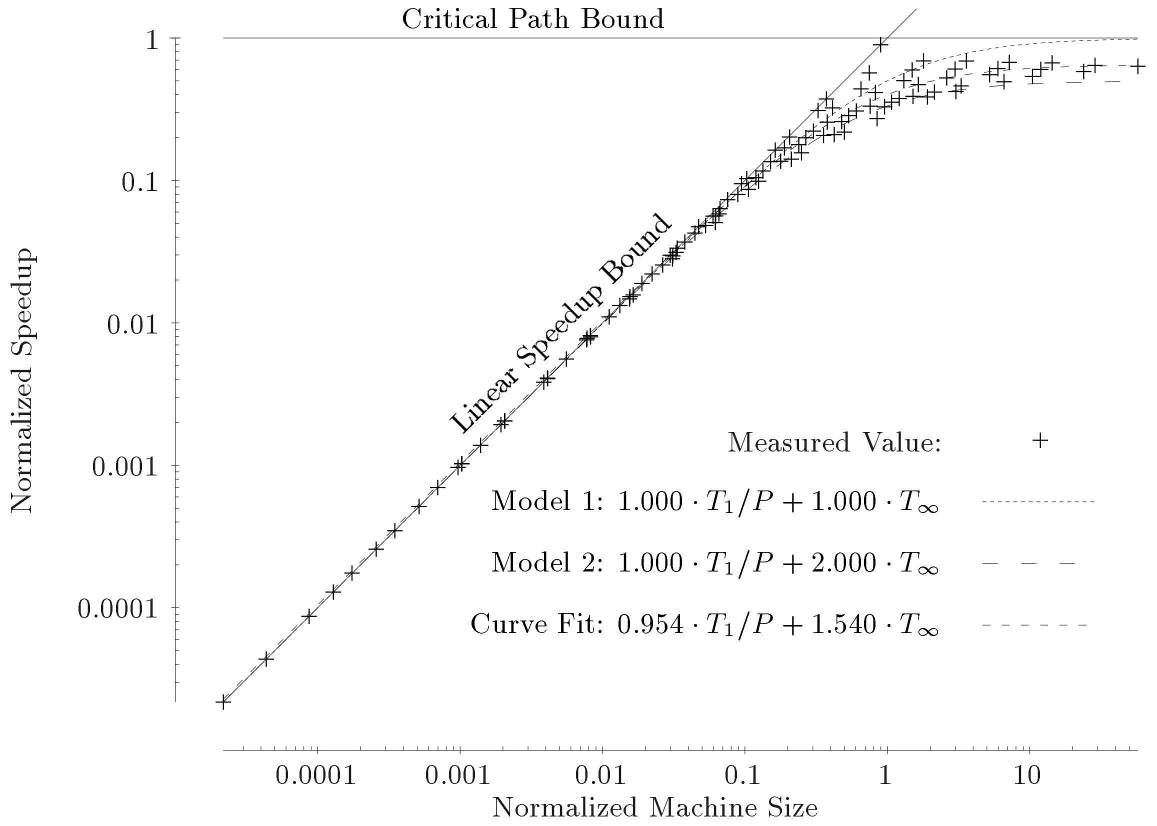



**Figure 5.14**: Normalized speedups for the `knary` synthetic benchmark using from 1 to 256 processors. The horizontal axis is the number $P$ of processors and the vertical axis is the speedup $T_1/T_P$, but each data point has been normalized by dividing by $T_1/T_\infty$.

ways within a factor of 4 of the critical-path upper bound.

The theoretical results from Section 5.3 show that the expected running time of a Cilk computation on $P$ processors is $T_P = O(T_1/P + T_\infty)$. Thus, it makes sense to try to fit the `knary` data to a curve of the form $T_P = c_1(T_1/P) + c_\infty(T_\infty)$. A least-squares fit to the data to minimize the relative error yields $c_1 = 0.9543 \pm 0.1775$ and $c_\infty = 1.54 \pm 0.3888$ with 95 percent confidence. The $R^2$ correlation coefficient of the fit is 0.989101, and the mean relative error is 13.07 percent. The curve fit is shown in Figure 5.14, which also plots the simpler curves $T_P = T_1/P + T_\infty$ and $T_P = T_1/P + 2 \cdot T_\infty$ for comparison. As can be seen from the figure, little is lost in the linear speedup range of the curve by assuming that the coefficient $c_1$ on the $T_1/P$ term equals 1. Indeed, a fit to $T_P = T_1/P + c_\infty(T_\infty)$ yields $c_\infty = 1.509 \pm 0.3727$ with $R^2 = 0.983592$ and a mean relative error of 4.04 percent, which is in some ways better than the fit that includes a $c_1$ term. (The $R^2$ measure is a little worse, but the mean relative error is much better.)

It makes sense that the data points become more scattered when $P$ is close to or exceeds the average parallelism. In this range, the amount of time spent in work stealing becomes

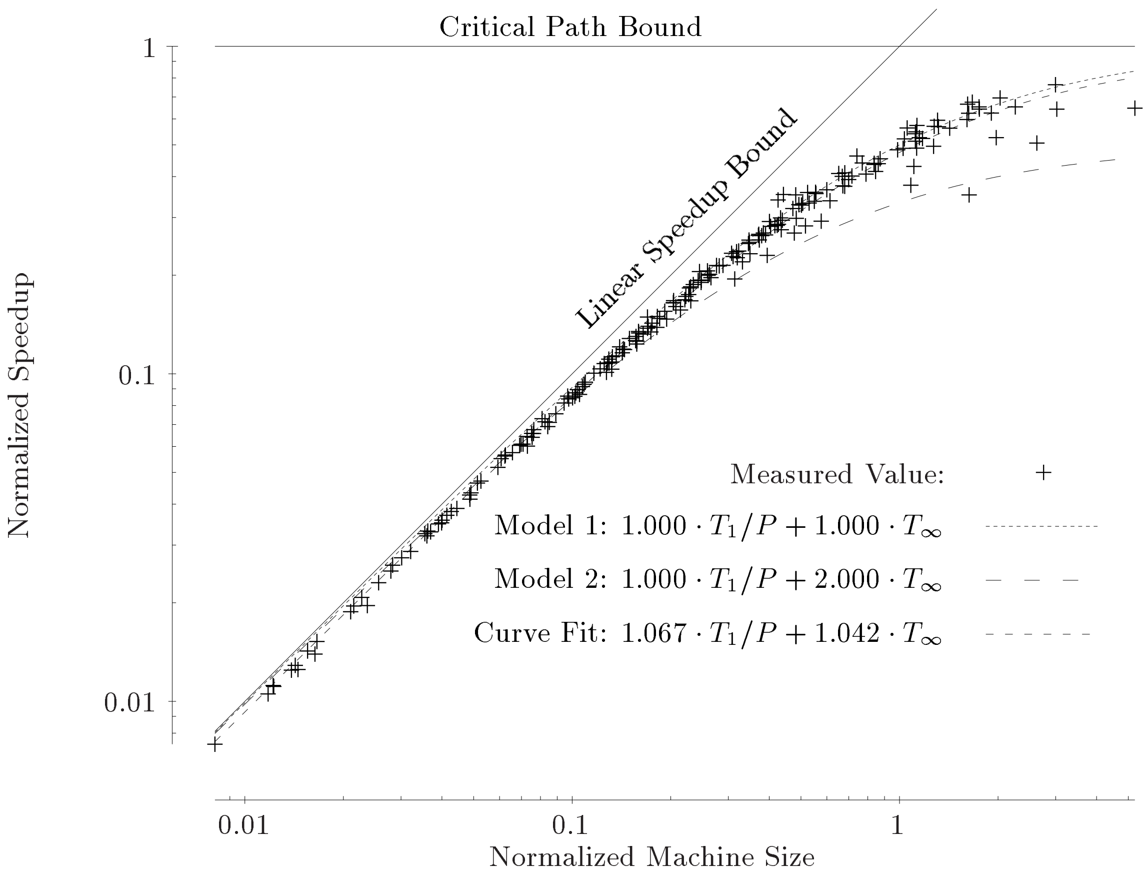



**Figure 5.15**: Normalized speedups for the ⋆Socrates chess program.

a significant fraction of the overall execution time. The real measure of the quality of a scheduler is how much larger than $P$ the average parallelism $T_1/T_\infty$ must be before $T_P$ shows substantial influence from the critical path. One can see from Figure 5.14 that if the average parallelism exceeds $P$ by a factor of 10, the critical path has almost no impact on the running time.

To confirm our simple model of the Cilk scheduler's performance on a real application, we ran ⋆`Socrates` on a variety of chess positions using various numbers of processors. Figure 5.15 shows the results of our study, which confirm the results from the `knary` synthetic benchmark. The best fit to $T_P = c_1(T_1/P) + c_\infty(T_\infty)$ yields $c_1 = 1.067 \pm 0.0141$ and $c_\infty = 1.042 \pm 0.0467$ with 95 percent confidence. The $R^2$ correlation coefficient of the fit is 0.9994, and the mean relative error is 4.05 percent.

By using work and critical path length to model the performance of an application under development, we can avoid being trapped by the following interesting anomaly. After making an "improvement" to the program, we find that, in test runs on a small-scale parallel machine, the program runs faster. Lacking any other information, we may conclude

| original program | "improved" program |
|---|---|
| $T_1 = 2048$ seconds | $T_1 = 1024$ seconds |
| $T_\infty = 1$ second | $T_\infty = 8$ seconds |

$$T_{32} = T_1/32 + T_\infty$$

| | |
|---|---|
| $T_{32} = 2048/32+1$ | $T_{32} = 1024/32+8$ |
| $= 65$ seconds | $= 40$ seconds |

$$T_{512} = T_1/512 + T_\infty$$

| | |
|---|---|
| $T_{512} = 2048/512+1$ | $T_{512} = 1024/512+8$ |
| $= 5$ seconds | $= 10$ seconds |

**Figure 5.16**: A speedup anomaly. We use the simple model $T_P = T_1/P + T_\infty$. On 32 processors, the "improved" program runs 38 percent faster than the original, taking 40 seconds compared to the original's 65 seconds. But it does so with less work at the cost of a longer critical path, and on 512 processors, the "improved" program is actually twice as slow as the original.

that the "improved" program is indeed superior to the original. But by measuring work and critical path length, we model our program's performance and predict that, on a large-scale machine, the "improved" program is actually slower than the original. Of course, we then confirm this prediction by performing a test on the large machine.

This speedup anomaly occurs because the "improved" program does less work at the cost of a longer critical path. Figure 5.16 illustrates this phenomenon with synthetic numbers. In this example, the "improved" program runs 38 percent faster than the original (40 seconds versus 65 seconds) on 32 processors. But on 512 processors, the "improved" program is actually twice as slow as the original.

Indeed, as some of us were developing and tuning heuristics to increase the performance of `⋆Socrates`, we used work and critical path length as our measures of progress. At that time, our platform for competition was the entire 512-node CM5 in dedicated mode at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. Unable to use this machine in our day-to-day work, we did development and testing on a 32-node time-shared partition of a CM5 that we have in house. More than once, by using work and critical path measurements taken from our 32-processor runs to predict the runtime of our program on the 512-processor machine, we avoided falling into the trap of the speedup anomaly just described.

## 5.3 A theoretical analysis of the Cilk scheduler

Cilk's work-stealing scheduler executes any "fully strict" Cilk program using space, time, and communication all within a constant factor of optimal. In previous chapters, we proved analogous results using a model of multithreaded computation that is somewhat simpler than Cilk's model. Rather than use the more complex Cilk model in these previous proofs, we chose to use the simpler model in order to make the proofs more tractable and keep from obscuring the essential ideas. In this section, we show how these proofs must be modified to account for the Cilk model.

Recall that a Cilk computation models the execution of a Cilk program as a tree of procedures and a dag of threads that unfold dynamically during program execution. For analysis, we refine the dag of threads into a dag of unit-size instructions as follows. Each thread is broken into instructions connected by *continue* edges into a linear sequence from the first instruction of the thread to the last. In the example Cilk computation illustrated in Figure 5.17, thread $\tau_2$ contains 3 instructions: $v_5$, $v_6$, and $v_7$. If a thread $\tau$ spawns a child $\tau'$ with the `spawn` primitive, then the dag has a *spawn* edge from the appropriate instruction of $\tau$ to the first instruction of $\tau'$. Recall that the child thread $\tau'$ is the initial thread of a child procedure. In our example computation, thread $\tau_1$ spawns the child thread $\tau_2$ (the initial thread of procedure $\Gamma_2$) as represented by the spawn edge $(v_4, v_5)$. If $\tau$ spawns a successor $\tau'$ with the `spawn_next` primitive, then the dag has a *spawn-next* edge from the appropriate instruction of $\tau$ to the first instruction of $\tau'$. Recall that the successor thread $\tau'$ is in the same procedure as $\tau$. In our example computation, thread $\tau_1$ spawns the successor thread $\tau_9$ as represented by the spawn-next edge $(v_2, v_{22})$. Procedure $\Gamma_1$ contains 3 threads: $\tau_1$, $\tau_9$, and $\tau_{10}$. In general, the instructions are formed into threads by continue edges; the threads are formed into procedures by spawn-next edges; and the procedures are formed into a spawn tree by the spawn edges. In our example computation, the root procedure $\Gamma_1$ has 2 children, $\Gamma_2$ and $\Gamma_6$, and procedure $\Gamma_2$ has 3 children: $\Gamma_3$, $\Gamma_4$, and $\Gamma_5$. Procedures $\Gamma_3$, $\Gamma_4$, $\Gamma_5$, and $\Gamma_6$, which have no children, are *leaf* procedures. If a thread $\tau$ sends a value (or signal) to another thread $\tau'$ with the `send_argument` or `accumulate` statement, then the dag has a *dependency* edge from the appropriate instruction of $\tau$ to the first instruction of $\tau'$. In our example computation, thread $\tau_3$ sends a value (or signal) to $\tau_5$ as represented by the dependency edge $(v_9, v_{12})$. Notice that only an instruction that is the first instruction of its thread can have an incoming

Some of the research reported in this section is joint work with Charles Leiserson and Keith Randall both of MIT's Laboratory for Computer Science. The material in this section generalizes results previously published in [12].

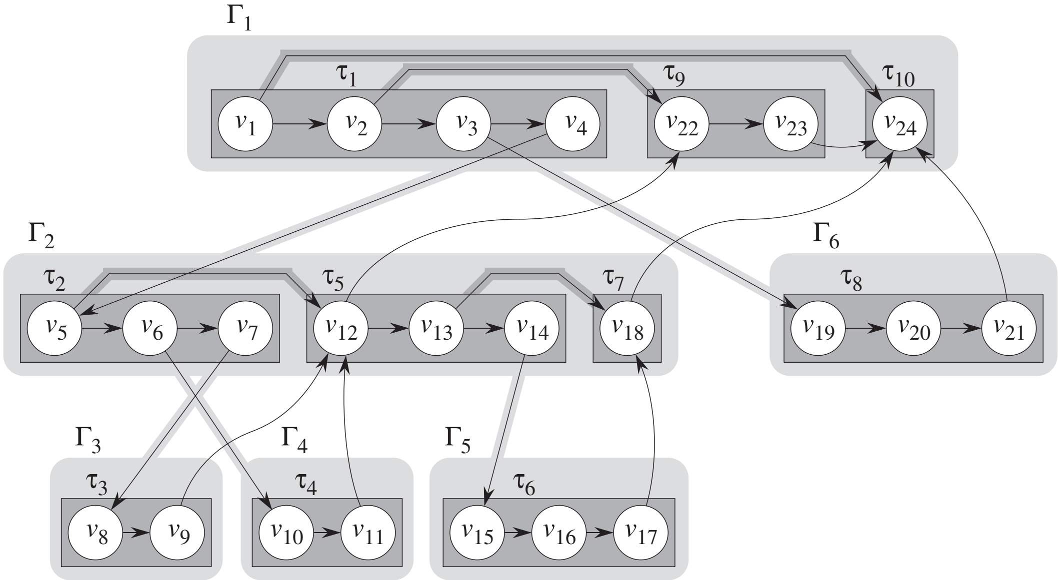



**Figure 5.17**: A Cilk computation. This computation contains 24 instructions $v_1, v_2, \ldots, v_{24}$ represented by the circles, and 10 threads $\tau_1, \tau_2, \ldots, \tau_{10}$ represented by the dark-shaded rectangles, and 6 procedures $\Gamma_1, \Gamma_2, \ldots, \Gamma_6$ represented by the light-shaded rounded rectangles. The continue edges are horizontal within a thread; the spawn-next edges are dark shaded within a procedure; the spawn edges are light shaded between procedures; and the dependency edges are curved between procedures. The spawn edges emerging from a thread cross each other so that the 1-processor execution order $v_1, v_2, \ldots, v_{24}$ proceeds from left to right. This figure does not show the "ghosts" required for the analysis.

edge that is not a continue edge. Consistent with our unit-time model of instructions, we assume that the out-degree of each instruction is at most some constant. The work $T_1$ is the number of instructions in the dag, and the critical path length $T_\infty$ is the length of a longest path in the dag. In our example computation, we have $T_1 = 24$ and $T_\infty = 17$.

Notice the difference between the 1-processor execution order $v_1, v_2, \ldots, v_{24}$ of our example Cilk computation (Figure 5.17) and the 1-processor left-to-right depth-first execution order $v_1, v_2, \ldots, v_{20}$ of the similar multithreaded computation shown in Figure 3.1 (page 33).

In order to discuss the execution of Cilk computations, we borrow and adapt some of the terminology developed in Chapter 2. When a thread is spawned, we allocate a closure for it, and we say the thread is *alive* or *living*. At any given time step during the execution, an instruction is *ready* if all of its predecessors in the dag have been executed, and a living thread is ready if its first instruction is ready. Ready threads can be executed by the scheduler. When the last instruction of a thread is executed, the thread *dies*, and we free its closure. When the initial thread of a procedure is spawned, we say the procedure is *alive* or *living*, and when the procedure no longer has any living threads, then the procedure *dies*.

A Cilk computation is *fully strict* if every dependency edge goes from a procedure to either itself or its parent procedure. Our example computation (Figure 5.17) is fully strict. Specifically, the fully strict condition requires that for every dependency edge $(v, v')$ where $v$ is an instruction in thread $\tau$ in procedure $\Gamma$ and $v'$ is an instruction in thread $\tau'$ in procedure $\Gamma'$, we have either $\Gamma' = \Gamma$—so $\tau'$ is a successor of $\tau$—or $\Gamma'$ is the parent of $\Gamma$—so $\tau'$ is a successor of the thread that spawned $\tau$ as a child. In other words, in a fully strict Cilk computation, threads only send values (or signals) to their successors or their parent thread's successors.

For any fully strict Cilk computation with work $T_1$ and critical path length $T_\infty$, and for any number $P$ of processors, we shall prove the following bounds on execution space, time, and communication.

- **Space:** The space used to execute the computation is at most $S_1 P$ where $S_1$ is the space used by a 1-processor execution of the Cilk computation.

- **Time:** The expected time to execute the computation is $O(T_1/P + n_l T_\infty)$ where $n_l$ is the maximum number of threads that any procedure can have simultaneously living in the computation.

- **Communication:** The expected number of bytes communicated during the execution is $O(P n_l T_\infty (n_d + S_{\max}))$ where $n_d$ is the maximum number of dependency edges between any pair of threads and $S_{\max}$ is the size of the largest closure in the computation.

The expected bounds on both time and communication can be converted to high-probability bounds with the addition of some small extra terms. The reader interested in the proofs of these bounds should be familiar with the proofs in Chapter 4 before proceeding.

The cornerstone in proving these results is a structural lemma (analogous to Lemma 4.1, page 45) that characterizes the procedures in the ready pool of any processor at any time. In Cilk, the ready pool actually contains threads, not procedures, but each thread belongs to a procedure, and we use the procedures to characterize the structure of the ready pool. In order to state and prove this structural lemma, we first need two technical assumptions and some new terminology.

Our first technical assumption, illustrated in Figure 5.18, is that any thread $\tau$ that spawns children also, as the last thing it does before dying, spawns a ready successor thread $\tau^*$. We call $\tau^*$ a *ghost thread*. A ghost thread contains only a single *ghost instruction* that takes no time to execute, so it is purely a technical convenience. Nevertheless, in our analysis, we shall assume that $\tau^*$ is handled by the scheduler just like any other thread. This ready successor thread simplifies the structural lemma. (The example computation of Figure 5.17 does not show the required ghost threads.)

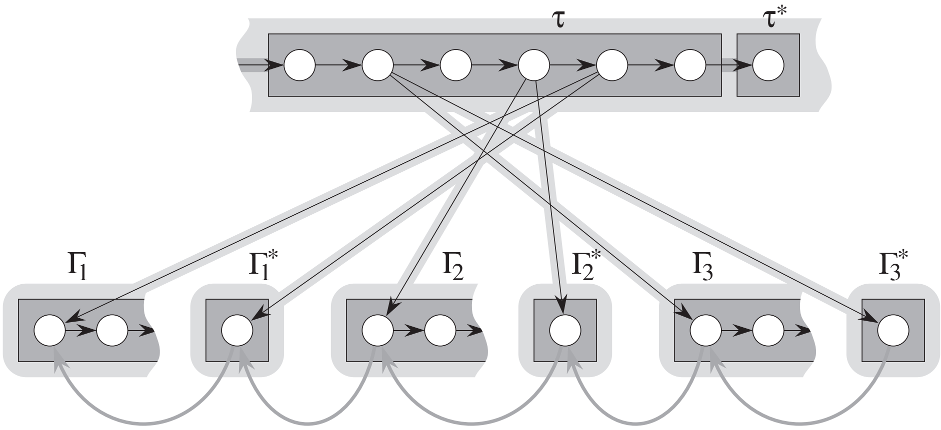



**Figure 5.18**: For each thread $\tau$ that spawns children, we augment the computation as follows. Thread $\tau$ spawns a successor ghost thread $\tau^*$ as the last thing it does. Each time $\tau$ spawns a child procedure $\Gamma$, it also spawns a ghost child $\Gamma^*$. The children are ordered from youngest to oldest going from left to right. The child procedures are linked by "pool edges" shown curved and gray.

Two procedures, $\Gamma_1$ and $\Gamma_2$ are *tuplets* if, in addition to having the same parent procedure, they also have the same parent thread. Since the instructions in a thread are totally ordered, the tuplet procedures can be ordered by age: the first child spawned is considered to be older than the second, and so on. In our example Cilk computation (Figure 5.17), the procedures $\Gamma_3$ and $\Gamma_4$ are tuplets with $\Gamma_4$ being the older of the two.

Our second technical assumption, illustrated in Figure 5.18, is that whenever a thread $\tau$ spawns a child procedure $\Gamma$, it also spawns a second child procedure $\Gamma^*$, called a *ghost procedure*. A ghost procedure contains only a ghost thread, and therefore, takes no time to execute. We think of the parent thread $\tau$ as spawning children in pairs $(\Gamma, \Gamma^*)$, with each pair having one ghost procedure. The scheduler treats $\Gamma^*$ (or rather its ghost thread) as having been spawned just before $\Gamma$, and therefore, the ghost is considered to be the older of the pair. Ghost procedures are purely a technical mechanism to facilitate our analysis. (The example computation of Figure 5.17 does not show the required ghost procedures.)

At any given time step during the execution and for any processor $p$, we define the *list of procedures at level $L$* as follows. We start with the *list of threads at level $L$*. Let $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ be the list of threads in the level $L$ list of $p$'s ready pool, ordered from $\tau_1$ at the head of the list to $\tau_n$ at the tail. If processor $p$ is not executing a thread at level $L$, then the list of threads at level $L$ is $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$. If processor $p$ is executing a thread $\tau$ at level $L$, then the list of threads at level $L$ is $\langle \tau, \tau_1, \tau_2, \ldots, \tau_n \rangle$. The list of procedures at level $L$ is the list $\langle \Gamma_1, \Gamma_2, \ldots, \Gamma_k \rangle$ of procedures derived from the list of threads by replacing each thread with its procedure and collapsing adjacent equal entries into one entry. We say that there are $k$ procedures at level $L$. As shorthand, we shall use the notation $\langle \Gamma_i \rangle$ to denote the

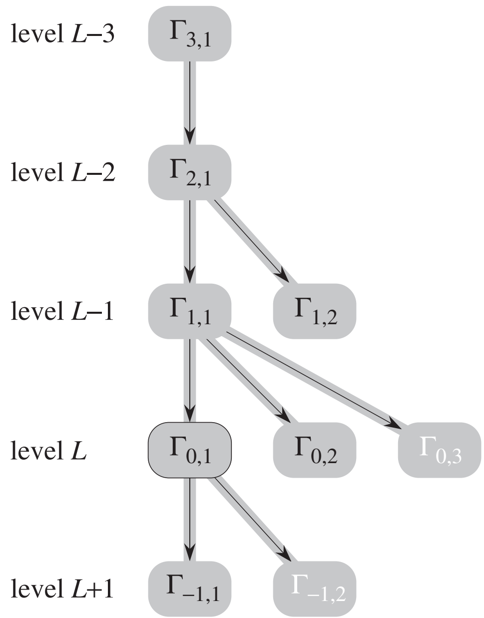



**Figure 5.19**: The structure of procedures in a processor's ready pool. The processor $p$ is executing a thread of procedure $\Gamma_{0,1}$ at level $L$. In this example, we have $l = 3$, so for each $i = 0, 1, 2, 3$, the procedures $\langle \Gamma_{i-1,j} \rangle$ are tuplet children of $\Gamma_{i,1}$. The procedures $\Gamma_{0,3}$ and $\Gamma_{-1,2}$ must be ghost procedures.

list $\langle \Gamma_1, \Gamma_2, \ldots, \Gamma_k \rangle$, and when we use double subscripts, we shall use the notation $\langle \Gamma_{i,j} \rangle$ to denote the list $\langle \Gamma_{i,1}, \Gamma_{i,2}, \ldots, \Gamma_{i,k_i} \rangle$ for a given value of $i$.

We now state and prove the lemma characterizing the structure of procedures in the ready pool of any processor during the execution of a fully strict Cilk computation. This lemma is the Cilk analog of the structural lemma (Lemma 4.1, page 45) in Chapter 4 which applied to our simpler call-return model of multithreaded computation. Figure 5.19 illustrates the lemma.

**Lemma 5.1** *During the execution of any fully strict Cilk computation, consider any processor p and any given time step at which p executes an instruction of a thread $\tau$. Let $\Gamma_{0,1}$ be $\tau$'s procedure, and let L be $\tau$'s level. For any i, let $\langle \Gamma_{i,1}, \Gamma_{i,2}, \ldots, \Gamma_{i,k_i} \rangle$ denote the list of procedures at level $L - i$ where $k_i$ is the number of procedures at level $L - i$. Let l be the largest integer such that $k_l > 0$ holds. Then these lists of procedures satisfy the following four properties:*

① *For $i = 0, 1, \ldots, l$, we have $k_i > 0$; for $i = -1$, we have $k_i \geq 0$; and for $i < -1$, we have $k_i = 0$.*

② *For every i, the procedures $\langle \Gamma_{i,j} \rangle$ are tuplets ordered from youngest to oldest. For $i = 0, 1, \ldots, l$, the procedures $\langle \Gamma_{i-1,j} \rangle$ are children of $\Gamma_{i,1}$, and the procedures $\langle \Gamma_{-1,j} \rangle$*

*are children of thread $\tau$.*

③ *For every $i$, if we have $k_i > 1$, then for $j = 2, 3, \ldots, k_i$, procedure $\Gamma_{i,j}$ has never been worked on, and if we have $k_{-1} > 0$, then $\Gamma_{-1,1}$ also has never been worked on.*

④ *If $l > 1$ holds, then for $i = -1, 0, \ldots, l-3$, we have $k_i \neq 1$ and if $k_i \neq 0$, then $\Gamma_{i,k_i}$ is a ghost.*

*Proof:* The proof is a straightforward induction on execution time. Execution begins with the initial thread of the root procedure in some processor's ready pool and all other ready pools empty, so the lemma vacuously holds at the outset. Now, consider any step of the execution at which processor $p$ executes an instruction from thread $\tau$ in procedure $\Gamma_{0,1}$ at level $L$. Let $\langle \Gamma_{i,1}, \Gamma_{i,2}, \ldots, \Gamma_{i,k_i} \rangle$ denote the list of procedures at level $L-i$; let $l$ be the largest integer such that $k_l > 0$ holds; and assume that all four properties hold. Let $\tau'$ denote the thread (if any) being worked on by $p$ after the step; let $\Gamma'_{0,1}$ be its procedure and $L'$ its level; let $\langle \Gamma'_{i,1}, \Gamma'_{i,2}, \ldots, \Gamma'_{i,k'_i} \rangle$ denote the list of procedures at level $L'-i$ after the step; and let $l'$ be the largest integer such that after the step, $k'_{l'} > 0$ holds. The proof consists of looking at the actions of the Cilk scheduler and showing that all four properties hold after the step.

Before looking at the Cilk scheduler's actions, we first review the things that a thread in a fully strict computation may do that cause scheduler action. Thread $\tau$ may spawn a child with Cilk's `spawn` primitive. (Recall that when a thread spawns a child, we also spawn a ghost, so spawns occur in pairs.) Thread $\tau$ may enable a successor thread by either performing a `spawn_next` with no missing arguments or by performing a `send_argument` or `accumulate` to a previously spawned successor and having the successor's join counter decrement to zero. Thread $\tau$ may enable a thread in its parent procedure by performing a `send_argument` or `accumulate` as just described. For fully strict computations, a thread $\tau$ may only enable threads in its own procedure or its parent procedure. These enabling activities all cause the scheduler to post a ready thread to the ready pool. Thread $\tau$ dies when it executes its last instruction, and this event causes the scheduler to start another iteration of its loop. We now examine these scheduler actions individually and show that they each preserve the four properties of the lemma.

If the thread $\tau$ spawns a pair of children, then $p$ posts the children at the head of the ready pool's level $L+1$ list. In this case, only the list of procedures at level $L+1$ changes. Specifically, we have $k'_{-1} = k_{-1} + 2$ with $\Gamma'_{-1,1}$ and $\Gamma'_{-1,2}$ being the new child procedures. Procedure $\Gamma'_{-1,2}$ is the ghost and is considered to be older than $\Gamma'_{-1,1}$. Also, if $k'_{-1} > 2$ holds, then we have $\Gamma'_{-1,j} = \Gamma_{-1,j-2}$ for $j = 3, 4, \ldots, k'_{-1}$. Now we can check that the four properties still hold. The first property does not apply to the level $L+1$ list, so we only check the other three. Property ②: The procedures $\langle \Gamma'_{-1,j} \rangle$ must be tuplet children, ordered from youngest to oldest, of thread $\tau$ in procedure $\Gamma'_{0,1} = \Gamma_{0,1}$, because before the spawn, the procedures $\langle\Gamma_{-1,j}\rangle$ are tuplet children of $\tau$ ordered from youngest to oldest. Property ③: None of the procedures $\langle\Gamma'_{-1,j}\rangle$ have ever been worked on. Property ④: We have $k'_{-1} > 1$ because $\tau$ just spawned 2 children. Moreover, if $l' > 1$ holds, then $\Gamma'_{-1,k'_{-1}}$ is a ghost, because before the spawn we have $l > 1$ which means that if $k_{-1} \neq 0$ holds, then $\Gamma_{-1,k_{-1}}$ is a ghost. In this case, $\Gamma'_{-1,k'_{-1}} = \Gamma_{-1,k_{-1}}$ is a ghost. Otherwise, we have $k'_{-1} = k_{-1} + 2 = 2$, so $\Gamma'_{-1,k'_{-1}} = \Gamma'_{-1,2}$ is the newly spawned ghost.

If the thread $\tau$ enables a successor thread, then $p$ posts the successor at the head of the ready pool's level $L$ list. In this case, the list of procedures at level $L$ does not change, because the successor thread is part of the same procedure $\Gamma_{0,1}$ as $\tau$, and this procedure is already at the head of the list. With no change in any of the lists of procedures, all properties continue to hold.

If the thread $\tau$ enables a thread in its parent procedure, then $p$ posts this newly enabled thread at the head of the ready pool's level $L-1$ list. If we have $l \geq 1$, then the parent procedure $\Gamma_{1,1}$ is already at the head of the level $L-1$ list, and therefore, none of the lists of procedures change. On the other hand, if we have $l = 0$, then we update $l' = l + 1 = 1$ with $k'_1 = 1$, and the list of procedures at level $L-1$ is the list $\langle\Gamma'_{1,1}\rangle$ containing only the parent procedure of $\Gamma'_{0,1} = \Gamma_{0,1}$. In this case, the properties are easily checked. Property ①: We have $k'_1 = 1 > 0$. Property ②: The procedures $\langle\Gamma'_{0,j}\rangle$ are children of $\Gamma'_{1,1}$, because $\Gamma'_{0,1}$ is a child of $\Gamma'_{1,1}$ and the procedures $\langle\Gamma'_{0,j}\rangle = \langle\Gamma_{0,j}\rangle$ are tuplets. Property ③: This property does not apply to the level $L-1$ list, because we have $k'_1 = 1$. Property ④: Again, this property does not apply, because we have $l' = 1$.

If the thread $\tau$ dies, then $p$ starts another iteration of the scheduling loop. If the ready pool is empty, then it commences work stealing, and the properties hold vacuously. Otherwise, it removes and executes the thread at the bottommost nonempty level of the ready pool. We consider two cases depending on whether the level $L+1$ list is nonempty.

If the level $L+1$ list is nonempty ($k_{-1} > 0$), then $p$ removes and executes the thread at the head of the ready pool's level $L+1$ list. Recall that because thread $\tau$ of procedure $\Gamma_{0,1}$ spawned children, it also spawns a ready successor thread $\tau^*$ as the last thing it does before dying. Therefore, the ready pool's list at level $L$ must have a thread of $\Gamma_{0,1}$ at its head when $\tau$ dies. We have $L' = L + 1$ and $l' = l + 1$. We also have $k'_{-1} = 0$, and for $i = 0, 1, \ldots l'$, we have $k'_i = k_{i-1}$ with $\Gamma'_{i,j} = \Gamma_{i-1,j}$ for $j = 1, 2, \ldots, k'_i$. We now check that the properties still hold. Properties ① and ②: We haven't actually changed any of the lists of procedures, only renamed them. Property ③: No procedure $\Gamma'_{i,j}$ with $j > 1$ has ever been worked on. Also, we have $k'_{-1} = 0$. Property ④: This property only applies to the level $L'$ list if we have $l' > 2$. If $l' > 2$ holds, then we have $k'_0 > 1$ and $\Gamma'_{0,k'_0}$ is a ghost, because before this step $l > 1$ holds, which means that we have $k_{-1} > 1$ and $\Gamma_{-1,k_{-1}}$ is a ghost.

Now, suppose that the level $L+1$ list is empty when thread $\tau$ dies. We further break this situation down into two cases depending on whether the ready pool's level $L$ list is nonempty. If the level $L$ list is nonempty, then $p$ removes and executes the thread at the head of this list. If this thread is in procedure $\Gamma_{0,1}$, then no list of procedures changes and the properties continue to hold. Otherwise, we must have $k_0 > 1$, and we now have $k'_0 = k_0 - 1$ and $\Gamma'_{0,j} = \Gamma_{0,j+1}$ for $j = 1, 2, \ldots, k'_0$. If $\Gamma'_{0,1}$ is a ghost, then we execute it immediately and advance to the next iteration of the scheduling loop. Therefore, we only need to check that the properties still hold in the case that $\Gamma'_{0,1}$ is not a ghost. The first three properties are easily checked and they continue to hold. Property ④: This property only applies to the level $L' = L$ list if we have $l' = l > 2$. If $l' > 2$ holds, then we have $k'_0 > 1$ and $\Gamma'_{0,k'_0}$ is a ghost, because before this step we have $l > 2$ and $\Gamma_{0,k_0}$ is a ghost, which means that we must have $k_0 > 2$ or else $\Gamma'_{0,1} = \Gamma_{0,2}$ would be a ghost.

If the level $L$ list is empty but the ready pool as a whole is nonempty, then the ready pool's level $L-1$ list must be nonempty, because otherwise we would violate Property ①. In this case, $p$ removes and executes the thread at the head of the level $L-1$ list. We now have $L' = L-1$ and $l' = l-1$. We also have $k'_{-1} = 0$, and for $i = 0, 1, \ldots, l'$ we have $k'_i = k_{i+1}$ with $\Gamma'_{i,j} = \Gamma_{i+1,j}$ for $j = 1, 2, \ldots, k'_i$. In this case, all four properties are easily checked and they continue to hold.

Finally, if some other processor steals a thread from processor $p$, then it removes the thread from the tail of the topmost nonempty list of $p$'s ready pool. Suppose that we have $l > 0$. In this case, the stolen thread is in procedure $\Gamma_{l,k_l}$, and the list of procedures at level $L-l$ may be shortened by one. If so, then we have $k'_l = k_l - 1$, and if we also have $k'_l = 0$, then we update $l' = l-1$. In these cases, the properties are easily checked and they continue to hold. Now, suppose that we have $l = 0$. If $k_0 > 1$ also holds, then the stolen thread is in procedure $\Gamma_{0,k_0}$ and the properties continue to hold as before. On the other hand, if we have $k_0 = 1$, then the level $L$ list of procedures contains only $\Gamma_{0,1}$ and there are two possibilities to consider. If the ready pool's level $L$ list is nonempty, then the stolen thread is a thread of procedure $\Gamma_{0,1}$. In this case, the list of procedures at level $L$ does not change, and the properties continue to hold. On the other hand, if the ready pool's level $L$ list is empty, then the stolen thread will be a thread from procedure $\Gamma_{-1,k_{-1}}$. In this case, the list of procedures at level $L+1$ shortens, but again, the properties continue to hold.

All other activity by processor $p$—such as work stealing or executing an instruction that does not invoke any of the above actions—clearly preserve the lemma. ∎

Our bound on space accounts for any stack-like memory. Specifically, we allow any instruction to allocate memory for its procedure provided that the instruction is totally ordered with respect to every other instruction in its procedure and provided that the instruction cannot be executed at a time step when its procedure has a living child. We allow any instruction to deallocate memory for its procedure provided that the instruction is totally ordered with respect to every other instruction in its procedure. Additionally, we require that all memory allocated for a procedure is deallocated before the procedure dies. At any given time step during the execution, the amount of memory currently allocated for a given living procedure is the net memory allocated for the procedure by all instructions that have been executed.

The space bound follows from the "busy-leaves" property which characterizes the living procedures at all time steps during the execution. At any given time step during the execution, we say that a procedure is a leaf if it has no living children, and we say that a leaf procedure is a *primary leaf* if, in addition, either it has no younger tuplets living or it has been worked on by some processor. The *busy-leaves* property states that every primary leaf procedure has a processor either working on it or working on its parent thread. To prove the space bound, we show that Cilk's scheduler maintains the busy-leaves property, and then we show that the busy-leaves property implies the space bound.

**Theorem 5.2** *For any fully strict Cilk computation, if $S_1$ is the space used to execute the computation on* 1 *processor, then with any number $P$ of processors, Cilk's work-stealing scheduler uses at most $S_1 P$ space.*

*Proof:* We first show that Cilk's work-stealing scheduler maintains the busy-leaves property, and then we show that the busy-leaves property implies the space bound.

To see that Cilk's scheduler maintains the busy-leaves property, consider any time step during the execution and any primary leaf procedure $\Gamma$. Since $\Gamma$ is a leaf and the computation is strict, there must be some thread in $\Gamma$ that is ready. Therefore, $\Gamma$ is in the list of procedures at some level of some processor $p$'s ready pool. Suppose processor $p$ is executing a thread $\tau$ at level $L$, and let $\langle \Gamma_{i,j} \rangle$ denote the list of procedures at level $L-i$. From Lemma 5.1, we know that among all of these procedures, the only one that can be a primary leaf is either $\Gamma_{0,1}$ (if we have $k_{-1} = 0$) or $\Gamma_{-1,1}$ (otherwise). In the former case, we have $\Gamma = \Gamma_{0,1}$, and $p$ is executing a thread of $\Gamma$. In the latter case, we have $\Gamma = \Gamma_{-1,1}$, and $p$ is executing the thread that spawned $\Gamma$. In either case, procedure $\Gamma$ is busy.

The $P$-processor space bound, $S_1 P$, is obtained by showing that at all time steps $t$ during the execution, every living procedure can be associated with a primary leaf procedure and that the total space currently allocated for all procedures assigned to a given primary leaf is at most $S_1$. We assign a procedure to a primary leaf as follows. If the procedure is a primary leaf, then we assign it to itself. If the procedure is a leaf but it is not a primary leaf, then we assign it to the same primary leaf as its youngest tuplet. If the procedure is not a leaf, then we assign it to the same primary leaf as any one of its living children. The procedures assigned to any given primary leaf $\Gamma$ are $\Gamma$'s ancestors and older tuplets of $\Gamma$'s ancestors.

Now, consider any primary leaf procedure $\Gamma$. Since $\Gamma$ is busy, there must be a processor $p$ that is executing either a thread of $\Gamma$ or the thread that spawned $\Gamma$. Let $\tau$ denote the thread that $p$ is executing, and let $v$ denote the instruction that $p$ executes at this time step. For any other procedure $\Gamma'$ assigned to $\Gamma$, we claim that the amount of memory currently allocated for $\Gamma'$ in our $P$-processor execution is no more than the amount of memory allocated for $\Gamma'$ at the time step in the 1-processor execution when the processor executes instruction $v$. To verify this claim, consider the two possible relationships that $\Gamma'$ may have with $\Gamma$. Suppose $\Gamma'$ is an ancestor of $\Gamma$. Consider the set of allocating or deallocating instructions in $\Gamma'$ that have been executed in our $P$-processor execution. This set must be a superset of the instructions in $\Gamma'$ that have been executed in the 1-processor case, because the allocating and deallocating instructions are totally ordered with respect to every other instruction in the procedure. Moreover, those allocating or deallocating instructions (if any) that have been executed in our $P$-processor case that haven't been executed in the 1-processor case must, in fact, be deallocating instructions, since they must have been executed while $\Gamma'$ had a living child. Thus the memory allocated for $\Gamma'$ in our $P$-processor case is no more than the memory allocated for $\Gamma'$ in the 1-processor case. Now, suppose $\Gamma'$ is the older tuplet of one of $\Gamma$'s ancestors. In this case, since $\Gamma'$ is a leaf, we know that $\Gamma'$ has never been worked on. Likewise, in the 1-processor case, $\Gamma'$ must be alive but never worked on. Finally, consider $\Gamma$ itself. The set of allocating or deallocating instructions in $\Gamma$ that have been executed in our $P$-processor execution must be the same as those executed in the 1-processor case. Thus, for every procedure assigned to $\Gamma$, the amount of memory currently allocated for it in our $P$-processor execution is no more than the amount of memory allocated for it in the 1-processor case. Thus, the total space assigned to $\Gamma$ is at most $S_1$.

Since Cilk's scheduler keeps all primary leaves busy, with $P$ processors we are guaranteed that at every time step during the execution, at most $P$ primary-leaf procedures can be living. Every living procedure is assigned to one of these $P$ primary leaves, and the total space of the procedures assigned to a given primary leaf is at most $S_1$. Therefore, the total space of all living procedures is at most $S_1P$. ∎

In bounding execution time, we assume that the machine is an asynchronous parallel computer with $P$ processors, and its memory can be either distributed or shared. We further assume that concurrent accesses to the same data structure are serially queued by an adversary as in the atomic-access model of Section 4.2. Specifically, if a processor attempts to steal a closure from a victim processor and no other thief is attempting to steal from the same victim, then the steal attempt—successful or not—takes one unit of time. If multiple thieves simultaneously attempt to steal from the same victim, then their requests are handled one per time step, in an order determined by an adversary.

In our analysis of execution time, we follow the same accounting argument as in Section 4.3. At each time step, we collect $P$ dollars, one per processor. At each time step, each processor places its dollar in one of three buckets according to its actions at that step. If the processor executes an instruction at the step, then it places its dollar into the WORK bucket. If the processor initiates a steal attempt at the step, then it places its dollar into the STEAL bucket. And, if the processor merely waits for a queued steal request at the step, then it places its dollar into the WAIT bucket. We shall derive the running time bound by bounding the number of dollars in each bucket at the end of the execution, summing these three bounds, and then dividing by $P$.

The bounds on the dollars in the WORK and WAIT buckets are exactly as in Section 4.3. Execution ends with $T_1$ dollars in the WORK bucket, since there are $T_1$ instructions in the computation (Lemma 4.4, page 53). Lemma 4.3 (page 50) bounds the number of dollars in the WAIT bucket as a function of the number of dollars in the STEAL bucket. With high probability, the number of dollars in the WAIT bucket is at most a constant times the number of dollars in the STEAL bucket.

To bound the dollars in the STEAL bucket, we use a delay-sequence argument very similar the one we used in Section 4.3, but we must modify this argument slightly to account for the Cilk model of multithreaded computation. As in Section 4.3, the work-steal attempts are partitioned into *rounds* of at least $3P$ but fewer than $4P$ consecutive steal attempts. Also as in Section 4.3, the delay sequence is defined in terms of an augmented dag $G'$ of instructions, obtained from the original dag $G$ by adding some new edges. For a Cilk computation, these new edges are called *pool edges*, and for each thread $\tau$ that spawns children, we add pool edges as follows. Let $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ denote the spawned child procedures, and recall that, we must have $n > 1$, since we require the spawns to occur in pairs with ghost procedures. Then, as illustrated in Figure 5.18, for each $i = 2, 3, \ldots, n$, we add a pool edge from the first instruction of $\Gamma_i$ to the first instruction of $\Gamma_{i-1}$. If $T_\infty$ is the length of a longest path in $G$, then the longest path in the augmented dag $G'$ has length at most $2T_\infty$. At any given time step during the execution, an instruction is *critical* if all of its predecessors in $G'$ have been executed. If, during the course of the execution, a large number of steals are attempted, then we can identify a sequence of instructions—the delay sequence—in this augmented dag such that a large number of rounds of steal attempts were initiated while an instruction from the sequence was critical. We show that such delay sequences are unlikely to occur, because a critical instruction is unlikely to remain critical across a modest number of steal-attempt rounds.

The delay sequence is defined exactly as in Definition 4.5 (page 55). We repeat the definition here.

**Definition 5.3** *A* delay sequence *is a 3-tuple* $(U, R, \Pi)$ *satisfying the following conditions:*

- $U = (u_1, u_2, \ldots, u_L)$ *is a maximal directed path in* $G'$. *Specifically, for* $i = 1, 2, \ldots, L-1$, *the edge* $(u_i, u_{i+1})$ *belongs to* $G'$, *instruction* $u_1$ *has no incoming edges in* $G'$ *(instruction* $u_1$ *must be the first instruction of the initial thread of the root procedure), and instruction* $u_L$ *has no outgoing edges in* $G'$.

- $R$ *is a positive integer.*

- $\Pi = (\pi_1, \pi_2, \ldots, \pi_L)$ *is a partition of the integer* $R$.

*The delay sequence* $(U, R, \Pi)$ *is said to* occur *during an execution if for each* $i = 1, 2, \ldots, L$, *at least* $\pi_i$ *steal-attempt rounds occur while instruction* $u_i$ *is critical.*

The following lemma states that if a large number of steal attempts take place during an execution, then a delay sequence with large $R$ must occur. This lemma and its proof are identical to Lemma 4.6 (page 55).

**Lemma 5.4** *Consider the execution of a fully strict Cilk computation with critical path length* $T_\infty$ *on a computer with* $P$ *processors. If at least* $4P(2T_\infty + R)$ *steal attempts occur during the execution, then some* $(U, R, \Pi)$ *delay sequence must occur.* ■

We now establish that a critical instruction is unlikely to remain critical across a modest number of steal-attempt rounds. Specifically, we first show that if no procedure can ever have more than $n_l$ simultaneously living threads, then after $O(n_l)$ steal-attempt rounds, a critical instruction is very likely to be executed. The following lemma establishes facts analogous to those established in Lemma 4.7 (page 56) and Lemma 4.8 (page 56).

**Lemma 5.5** *Consider the execution of any fully strict Cilk computation on a parallel computer with* $P \geq 2$ *processors. If no procedure of the computation can ever have more than* $n_l$ *simultaneously living threads, then for any instruction* $v$ *and any number* $r \geq 12n_l$ *of steal-attempt rounds, the probability that* $r$ *rounds occur while* $v$ *is critical is at most* $e^{-r}$.

*Proof:* We first use our structural lemma, Lemma 5.1, to show that if instruction $v$ is critical at some time step, then it must be the ready instruction of a thread that either is currently being executed or is at the tail of a list near the top of some processor's ready pool. We then use this fact to establish the probabilistic bound.

Consider any time step at which instruction $v$ is critical. If some processor is executing $v$'s thread, then $v$ will be executed at this time step, and the lemma holds. Therefore, suppose that $v$ is the first instruction of a thread $\tau$ in procedure $\Gamma$, and $\tau$ is in the ready pool of a processor $p$. Since $v$ is critical, every one of $\Gamma$'s older tuplets must have been worked on at some earlier time step, so by Property ③ of Lemma 5.1, procedure $\Gamma$ must be at the tail of the list of procedures at some level. By the same reasoning, every nonempty list of procedures above this level must contain only one procedure. Then, Property ④ of Lemma 5.1 ensures that this level is at most 3 from the top. Thus, since each procedure can have at most $n_l$ simultaneously living threads, once processor $p$ has satisfied $3n_l$ work-steal requests, we are guaranteed that $v$ has been executed.

Mimicking the proof of Lemma 4.8, we observe that if instruction $v$ remains critical across $r$ steal-attempt rounds, then of the at least $(3r-(3n_l-1))P$ steal attempts initiated at least $3n_l$ time steps before $v$ is executed, fewer than $3n_l$ of them choose a particular processor $p$ as the victim. Letting the random variable $X$ denote the number of these steal attempts that do choose processor $p$, we bound the probability that $X$ is less than $3n_l$ by using a Chernoff bound [1] on the lower tail of a binomial distribution with mean $\mu$:

$$\Pr\{X < \mu - a\} \le e^{a^2/2\mu}$$

for any $a > 0$. In our case, we have $\mu = 3r - 3n_l + 1$, and to bound $\Pr\{X < 3n_l\}$, we have $a = \mu - 3n_l = 3r - 6n_l + 1$. Thus, the probability that $v$ remains critical across $r$ steal-attempt rounds is at most

$$\begin{aligned} \Pr\{X < 3n_l\} &\le \exp\left[(3r-6n_l+1)^2/2(3r-3n_l+1)\right] \\ &\le e^{-r} \end{aligned}$$

for $r \ge 12n_l$. ■

We now complete the delay-sequence argument and bound the total dollars in the STEAL bucket. The proof of the following lemma is nearly identical to the proof of Lemma 4.9 (page 58). The only change is that we use the probabilistic bound of Lemma 5.5 instead of the bound given by Lemma 4.8.

**Lemma 5.6** *Consider the execution of any fully strict Cilk computation with critical path length $T_\infty$ on a parallel computer with $P$ processors. If no procedure ever has more than $n_l$ simultaneously living threads, then for any $\varepsilon > 0$, with probability at least $1-\varepsilon$, the execution terminates with at most $O(P(n_l T_\infty + \lg(1/\varepsilon)))$ dollars in the* STEAL *bucket, and the* *expected number of dollars in this bucket is* $O(Pn_lT_\infty)$. ■

With bounds on all three buckets, we now state and prove the theorem that bounds the total execution time for a fully strict Cilk computation.

**Theorem 5.7** *Consider any fully strict Cilk computation with work* $T_1$ *and critical path length* $T_\infty$ *such that no procedure can ever have more than* $n_l$ *simultaneously living threads. With any number* $P$ *of processors, Cilk's work-stealing scheduler runs the computation in expected time* $O(T_1/P + n_lT_\infty)$. *Moreover, for any* $\varepsilon > 0$, *with probability at least* $1-\varepsilon$, *the execution time on* $P$ *processors is* $O(T_1/P + n_lT_\infty + \lg P + \lg(1/\varepsilon))$.

*Proof:* Add up the dollars in the three buckets and divide by $P$. ■

The next theorem bounds the total amount of communication that a Cilk computation performs in a distributed model. The analysis assumes that at most a constant number of bytes need to be communicated to perform a `send_argument` or `accumulate` in the case when the join counter does not go to zero. In the case when the join counter does go to zero, then an entire closure may need to be communicated.

**Theorem 5.8** *Consider any fully strict Cilk computation with work* $T_1$ *and critical path length* $T_\infty$ *such that no procedure ever has more than* $n_l$ *simultaneously living threads. With any number* $P$ *of processors, the total number of bytes communicated by Cilk's work-stealing scheduler has expectation* $O(Pn_lT_\infty(n_d + S_{max}))$, *where* $n_d$ *is the maximum number of dependency edges between any pair of threads and* $S_{max}$ *is the size in bytes of the largest closure in the computation. Moreover, for any* $\varepsilon > 0$, *the probability is at least* $1-\varepsilon$ *that the total communication incurred is* $O(P(n_lT_\infty + \lg(1/\varepsilon))(n_d + S_{max}))$.

*Proof:* We prove the bound for the expectation. The high-probability bound is analogous. By our bucketing argument, the expected number of steal attempts is at most $O(Pn_lT_\infty)$. When a thread is stolen, the communication incurred is at most $S_{\max}$. We also have communication when a processor executing a thread performs a `send_argument` or `accumulate` to a thread whose closure is on another processor. If the join counter does not go to zero, then the amount of communication is $O(1)$, and this event can occur at most $n_d$ times per steal. If the join counter does go to zero, then the amount of communication is at most $S_{\max}$, and this event can occur at most once per steal. Thus, the expected total communication cost is $O(Pn_lT_\infty(n_d + S_{\max}))$. ■

# Chapter 6

# Cilk on a network of workstations

In order to execute Cilk programs efficiently on a network of workstations, the Cilk-NOW runtime system implements "adaptive parallelism" and transparent fault tolerance. *Adaptive parallelism* allows a Cilk application to run on a set of workstations that may grow and shrink dynamically during program execution. When a given workstation is not being used by its owner, the workstation automatically joins in and helps out with the execution of a Cilk program. When the owner returns to work, the machine automatically retreats from the Cilk program. Fault tolerance allows a Cilk application to continue execution even in the face of individual workstation crashes or reboots. Applications may take advantage of this feature despite being *fault oblivious*. The application is written as an ordinary Cilk program with no special provision for handling faults. Recently, we ran the Cilk protein-folding application `pfold` (see page 78) using Cilk-NOW on a network of about 50 Sun SparcStations connected by Ethernet to solve a large-scale protein-folding problem. The program ran for 9 days, surviving several machine crashes and reboots, utilizing 6566 processor-hours of otherwise-idle cycles, with no administrative effort on our part (besides typing `pfold` at the command-line to begin execution), while other users of the network went about their business unaware of the program's presence. In this chapter, we show how the Cilk-NOW runtime system leverages the structure in Cilk's programming model to implement adaptive parallelism and fault tolerance. In Section 6.1, we present the architecture of Cilk-NOW. Then in Sections 6.2 and 6.3, we present the implementation of adaptive parallelism and fault tolerance.

---

Some of the research reported in this chapter is joint work with David Park formerly of MIT's Laboratory for Computer Science and now of McKinsey & Company and Phil Lisiecki of MIT's Laboratory for Computer Science. Some of the material in this chapter was previously published in [15]. At that time, Cilk-NOW was in the prototype stage and was called *Phish*.

## 6.1 System architecture

The Cilk-NOW runtime system consists of several component programs that work together recruiting idle machines in the network to work on the execution of Cilk programs, scheduling these idle machines among all the Cilk programs that are running, and managing the execution of each individual Cilk program. In this section, we shall cover the architecture of the Cilk-NOW runtime system, explaining the operation of each component program and their interactions.

In Cilk-NOW terminology, we refer to an executing Cilk program as a Cilk *job*. Since Cilk programs are parallel programs, a Cilk job consists of several processes running on several machines. One process, called the *clearinghouse*, in each Cilk job runs a program called `CilkChouse`. `CilkChouse` is a system-supplied program that is responsible for keeping track of all the other processes that comprise a given job. These other processes are called *workers*. A worker is a process running the actual executable of a Cilk program such as `ray` or `pfold`. Since Cilk jobs are adaptively parallel, the set of workers is dynamic. At any given time during the execution of a job, a new worker may join the job or an existing worker may leave. Thus, each Cilk job consists of one or more workers and a clearinghouse to keep track of them.

The Cilk-NOW runtime system contains two additional components to keep track of the Cilk jobs and the individual workstations in the network. The *job broker* is a processes running a system-supplied program called `CilkJobBroker`. The job broker runs on a single machine in the network and keeps track of the set of Cilk jobs running in the network. A *node manager* is a process running a system-supplied program called `CilkNodeManager`. A node manager runs as a background daemon on every machine in the network. It continually monitors its machine to determine when the machine is idle and when it is busy. When the node manager finds that its machine is idle, it contacts the job broker to find a running Cilk job that the machine can work on.

To see how all of these components work together in managing the execution of Cilk jobs, we shall run though an example. Suppose that the job broker is running on a machine called `Vulture`, and a user sits down at a machine called `Penguin` to run the `pfold` program. In our example, the user types

```
pfold 3 7
```

at the shell, thereby launching a Cilk job to enumerate all protein foldings using 3 initial folding sequences and starting with the 7th one.

The new Cilk job begins execution as illustrated in Figure 6.1. The new process running the `pfold` executable is the first worker and begins execution by forking a clearinghouse with the command line

```
CilkChouse -- pfold 3 7.
```

Thus, the clearinghouse knows that it is in charge of a job whose workers are running "`pfold 3 7`." The clearinghouse begins execution by sending a *job description* to the job broker. The job description is a record containing several fields. Among these fields is the name of the Cilk program executable—in this case `pfold`—and the clearinghouse's network address. The clearinghouse also sends its network address through a pipe back to the first worker, the worker that forked the clearinghouse. The clearinghouse then goes into a service loop waiting for messages from its workers. After receiving the clearinghouse's address from the pipe, the first worker *registers* with the clearinghouse by sending the clearinghouse a message containing its own network address. Now the clearinghouse knows about one worker, and it responds to that worker by assigning it a unique *name*. Workers are named with numbers, starting with number `0`. The first worker to register is named with number `0`, the second worker to register is named with number `1`, and so on. Having registered, worker `0` begins executing the Cilk program as described in the previous chapter. We now have a running Cilk job with one worker.

A second worker joins the Cilk job when some other workstation in the network becomes idle, as illustrated in Figure 6.2. Suppose the node manager on a machine named `Sparrow` detects that the machine is idle. The node manager sends a message to the job broker, informing the job broker of the idle machine. The job broker responds with the job description of a Cilk job for the machine to work on. In this case, the job description specifies our `pfold` job by giving the name of the executable—`pfold`—and the network address of the clearinghouse. The node manager then uses this information to fork a new worker as a child with the command line

```
pfold -NoChouse -Address=clearinghouse-address --.
```

The `-NoChouse` flag on the command line tells the worker that it is to be an additional worker in an already existing Cilk job. (Without this flag, the worker would fork a new clearinghouse and start a new Cilk job.) The `-Address` field on the command line tells the worker where in the network to find the clearinghouse. The worker uses this address to send a registration message, containing its own network address, to the clearinghouse. The clearinghouse responds with the worker's assigned name—in this case, number `1`—and the job's command-line arguments—in this case, "`pfold 3 7`." Additionally, the clearinghouse responds with a list of the network addresses of all other registered workers. Now the new

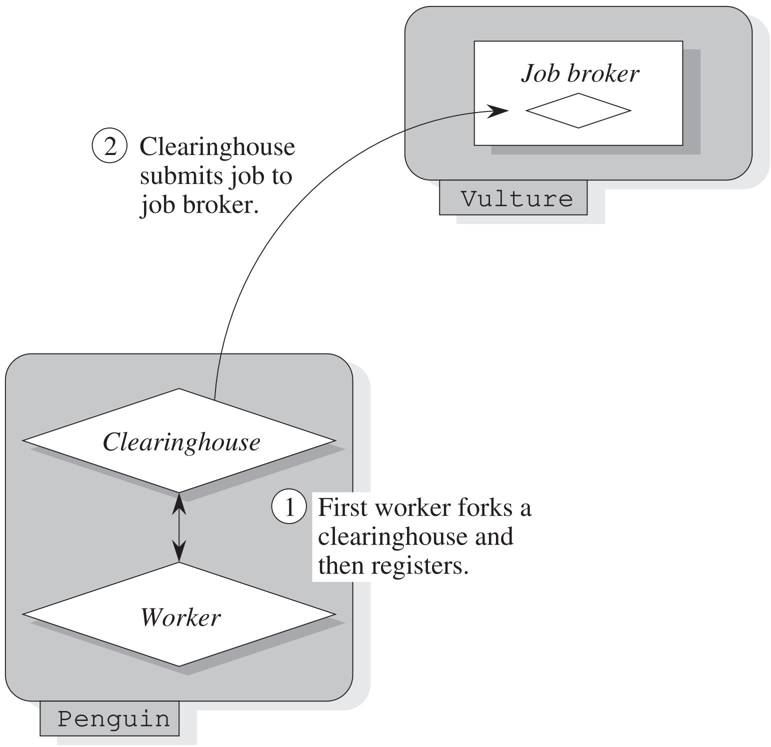



**Figure 6.1**: A Cilk job starts. The first worker forks a clearinghouse, and then the clearinghouse submits the job to the job broker.

worker knows the addresses of the other workers, so it can commence execution of the Cilk program and steal work as described in the previous chapter. We now have a running Cilk job with two workers.

Now, suppose that someone touches the keyboard on `Sparrow`. In this case, the node manager detects that the machine is busy, and the machine leaves the Cilk job as illustrated in Figure 6.3. After detecting that the machine is busy, the node manager sends a kill signal to its child worker. The worker catches this signal and prepares to leave the job. First, the worker offloads all of its closures to other workers as explained in more detail in the next section. Next, the worker sends a message to the clearinghouse to *unregister*. Finally, the worker terminates. When the node manager detects that its child worker has terminated, it notifies the job broker, so that the job broker can keep track of the number of workers in each Cilk job.

When a Cilk job is running, each worker checks in with the clearinghouse once every 2 seconds. Specifically, each worker, every 2 seconds, sends a message to the clearinghouse. The clearinghouse responds with an *update* message informing the worker of any other workers that have left the job and any new workers that have joined the job. For each new worker that has joined, the clearinghouse also provides the network address. If

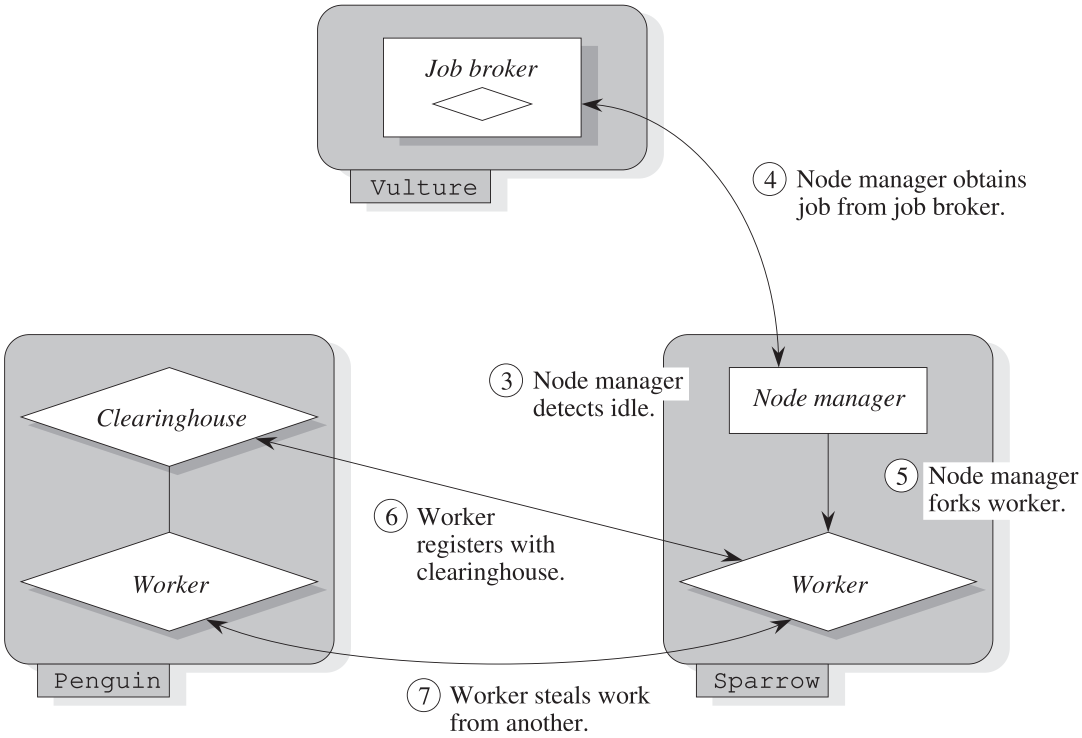



**Figure 6.2**: An idle machine joins a Cilk job. When the node manager detects that its machine is idle, it obtains a job from the job broker and then forks a worker. The worker registers with the clearinghouse and then begins work stealing.

the clearinghouse does not receive any messages from a given worker for over 30 seconds, then the clearinghouse determines that the worker has crashed. In later update messages, the clearinghouse informs the other workers of the crash, and the other workers take appropriate remedial action as described in Section 6.3. If the clearinghouse incorrectly determines that a worker has crashed and then receives a message from that worker, then the clearinghouse refuses to reply with update messages. A worker that receives no update messages from the clearinghouse for over 30 seconds commits suicide. Thus, a worker incorrectly determined to be crashed will eventually crash, and the communication protocols ensure correct operation in this event. With each worker communicating with the clearinghouse once to register, once to unregister, and once every 2 seconds for an update, we expect that a clearinghouse can service up to 1000 workers. Beyond this level and in a wide-area network, we may require multiple clearinghouses configured in a hierarchy.

All of the communication between workers and between workers and the clearinghouse is implemented with UDP/IP [95]. UDP/IP is an unreliable datagram protocol built on top of the internet protocol [25]. The protocols implemented in the Cilk-NOW runtime system all use UDP/IP to perform *split-phase* communication, so except in the case of work stealing, a worker never sits idle waiting for a reply or an acknowledgment. Knowing that UDP data-

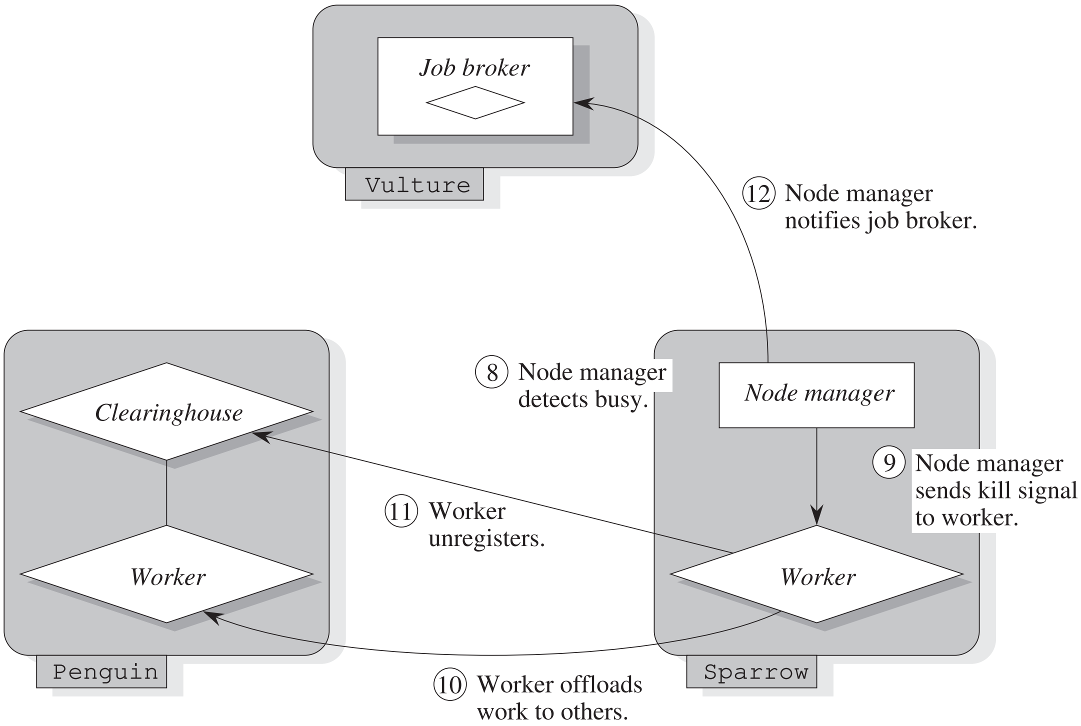



**Figure 6.3**: A no-longer-idle machine leaves a Cilk job. When the node manager detects that its machine is no-longer idle, it sends a kill signal to the worker. The worker catches this signal, offloads its work to other workers, unregisters with the clearinghouse, and then terminates.

grams are unreliable, the Cilk-NOW protocols incorporate appropriate mechanisms, such as acknowledgments, retries, and timeouts, to ensure correct operation when messages get lost. We shall not discuss these mechanisms in any detail, and in order to simplify our exposition of Cilk-NOW, we shall often speak—and indeed we already have spoken—of messages being sent and received as if they are reliable. What we will say about these mechanisms is that they are not built on top of UDP in any effort to create a reliable message-passing layer. Rather these mechanisms are built directly into the runtime system's split-phase protocols, so in the common case when a message does get through, Cilk-NOW pays no overhead to make the message reliable.

We chose to build Cilk-NOW's communication protocols using an unreliable message-passing layer instead of a reliable one for two reasons. First, reliable layers such as TCP/IP [95], PVM [96], and MPI [43] all perform implicit acknowledgments and retries to achieve reliability. Therefore, such layers either preclude the use of split-phase communication or require extra buffering and copying. A layer such as UDP which provides minimal service guarantees can be implemented with considerably less software overhead than a layer with more service features. In the common case when the additional service is not needed, the minimal layer can easily outperform its fully-featured counterpart. Second, in an environment where machines can crash and networks can break, the notion of a “reliable” message-passing layer is somewhat suspect. A runtime system operating in an inherently unreliable environment cannot expect the message-passing layer to make the environment reliable. Rather, the runtime system must incorporate appropriate mechanisms into its protocols to take action when a communication endpoint or link fails. For these reasons, we chose to build the Cilk-NOW runtime system on top of a minimal layer of message-passing service and incorporate mechanisms directly into the runtime system’s protocols in order to handle issues of reliability. The downside to this approach is complexity. The protocols implemented in the Cilk-NOW runtime system are complex: the code for these protocols takes almost 20 percent of the total runtime system code, and the programming effort was probably near half of the total. Nevertheless, this was a one-time effort that we expect will reap performance rewards for a long time to come.

The job broker, `CilkJobBroker`, and node manager, `CilkNodeManager`, are implemented using remote procedure calls (RPC) [7] in the standard client/server configuration with the job broker as the server. When the node manager on a machine finds that its machine is idle, it makes a remote procedure call to the job broker to obtain a Cilk job description. We shall finish this section by explaining the operation of the node manager and the job broker in their current incarnations.

Each machine in the network runs a node manager in the background. In general, when the machine is being used, the node manager wakes up every 5 seconds to determine if the machine has gone idle. It looks at how much time has elapsed since the keyboard and mouse have been touched, the number of users logged in, and the processor load averages. The node manager then passes these values through a predicate to decide if the machine is idle. This predicate can be customized for each machine. A typical predicate might require that the keyboard and mouse have not been touched for at least 2 minutes and the 1-minute processor load average is below 0.3. Alternatively, the owner of a machine might set the predicate to require that no users are logged in. We believe that maintaining the owner’s sovereignty is essential if we want owners to allow their machines to be used for parallel computation. When the predicate is satisfied, the machine is idle, and the node manager obtains a Cilk job description from the job broker and forks a worker. The node manager then monitors the worker and continues to monitor the machine. With a worker running, the node manager wakes up once every second to determine if the machine is still idle (adding 1.0 to any processor load-average threshold). If the machine is no longer idle, then the node manager sends a kill signal to the worker as previously described. When the worker process dies for any reason, the node manager takes one of two possible actions. If the machine is still idle, then it goes back to the job broker for another job to work on. If the machine is no longer idle, then it returns to monitoring the machine once every 5 seconds.

The job broker determines which idle machines work on which Cilk jobs. In its current implementation, the job broker uses a simple nonpreemptive, round-robin scheduling policy. This policy is extremely unfair in that it allows a single job to hold all the idle machines to the exclusion of all other jobs. For example, if one job is running and using all the idle machines, then when a second job starts, it will get none. Not until some machines leave the first job and then later get reassigned by the job broker, will machines join the second job. We are currently analyzing and experimenting with a simple probabilistic scheme based on random preemptive reassignments in order to obtain "fair" scheduling. Our proposed scheme uses the work-steal rates of individual workers to govern the reassignment probabilities. Each job should get its fair share of the idle machines, but no job should get more machines than it can efficiently utilize. In this proposed scheme, the functionality of the job broker is implemented in a totally distributed manner instead of in the current client/server manner.

## 6.2 Adaptive parallelism

Adaptive parallelism allows a Cilk job to take advantage of idle machines whether or not they are idle when the job starts and whether or not they will remain idle for the duration of the job. In order to efficiently utilize machines that may join and leave a running job, the overhead of supporting this feature must not excessively slow down the work of any worker at a time when it is not joining or leaving. As we saw in the previous section, a new worker joins a job easily enough by stealing a closure. A worker leaves a job by migrating all of its closures to other workers, and here the danger lies. When we migrate a waiting closure, other closures with continuations that refer to this closure must somehow update these continuations so they can find the waiting closure at its new location. (Without adaptive parallelism, waiting closures never move.) Naively, each migrated waiting closure would have to inform every other closure of its new location. In this section, we show how we can take advantage of strictness and the work-stealing scheduler to make this migration extremely simple and efficient.

Our approach is to impose additional structure on the organization of closures and continuations, such that the structure is cheap to maintain while simplifying the migration of closures. Specifically, we maintain closures in "subcomputations" that migrate en masse, and every continuation in a closure refers to a closure in the same subcomputation. In order to send a value from a closure in one subcomputation to a closure in another, we forward the value through intermediate "result closures," and give each result closure the ability to send the value to precisely one other closure in one other subcomputation. With this structure and these mechanisms, all of the overhead associated with adaptive parallelism (other than the actual migration of closures) occurs only when closures are stolen, and as we saw in Chapter 5, the number of steals grows at most linearly with the critical path of the computation but is not a function of the work. The bulk of this section's exposition concerns the organization of closures in subcomputations and the implementation of continuations. After covering these topics, the mechanism by which closures are migrated to facilitate adaptive parallelism is quite straightforward.

In Cilk-NOW, every closure is maintained in one of three pools associated with a data structure called a *subcomputation*. A subcomputation is a record containing (among other things) three pools of closures. The *ready pool* is the leveled lists of ready closures described in Section 5.1. The *waiting pool* is a list of waiting closures. The *assigned pool* is a list of ready closures that have been stolen away. Program execution begins with one subcomputation—the *root* subcomputation—allocated by worker `0` and containing a single closure—the initial thread of `cilk_main`—in the ready pool. In general, a subcomputation with any closures in its ready pool is said to be *ready*, and ready subcomputations can be executed by the scheduler as described in Section 5.1 with the additional provision that each waiting closure is kept in the waiting pool and then moved to the ready pool when its join counter decrements to zero. The assigned pool is used in work stealing as we shall now see.

When a ready closure is stolen from the ready pool of a victim worker's subcomputation, the closure is moved to the assigned pool, and *assigned* to a *thief subcomputation* newly allocated by the thief worker. The assignment is recorded by giving the thief subcomputation a unique *name* and storing that name in a record of *assignment information* attached to the assigned closure, as illustrated in Figure 6.4. The subcomputation's name is formed by concatenating the worker's name and a number unique to that worker. The first subcomputation allocated by a worker `r` is named `r:1`, the second is named `r:2`, and so on. The root subcomputation is named `0:1`. The thief subcomputation stores its own name and the name of its victim worker. The victim's assigned closure stores the name of the thief worker and the name of the thief subcomputation in its assignment information. We refer to the assigned closure as the thief subcomputation's *victim closure*. Thus, the victim closure and thief subcomputation can refer to each other via the thief subcomputation's name which is stored both in the victim closure's assignment information and in the thief subcomputation.

This link between a victim closure and a thief subcomputation is created during work stealing as follows. If a worker needs to steal work, then before sending a steal request to a

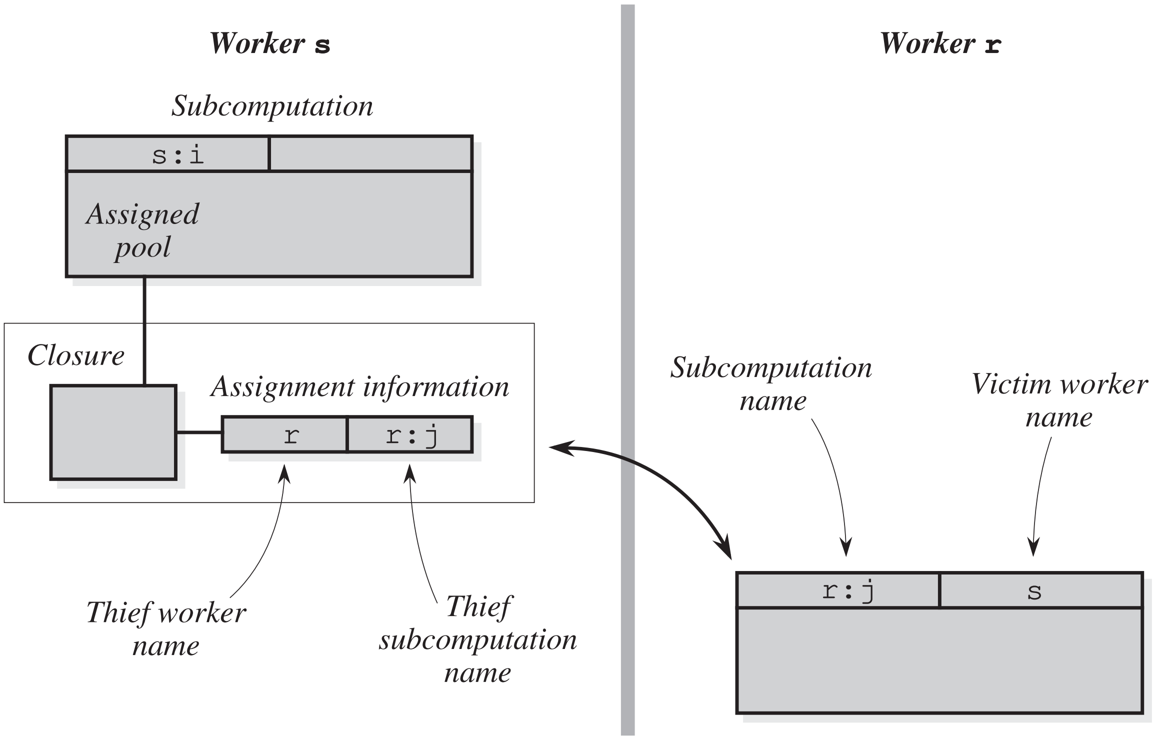



**Figure 6.4**: A victim closure stolen from the subcomputation `s:i` of victim worker `s` is assigned to the thief subcomputation `r:j`. The victim closure is placed in the assigned pool and augmented with assignment information that records the name of the thief worker and the name of the thief subcomputation. The thief subcomputation records its own name and the name of the victim worker. Thus, the victim closure and thief subcomputation can refer to each other via the thief subcomputation's name.

victim, it allocates a new thief subcomputation from a simple runtime heap. The thief subcomputation's name is contained in the steal request message. When the victim worker gets the request message, if it has any ready subcomputations, then it chooses a ready subcomputation in round-robin fashion, removes the closure at the tail of the topmost nonempty level in the subcomputation's ready pool, and places this victim closure in the assigned pool. The victim worker then assigns the closure to the thief subcomputation by adding to the closure an assignment information record allocated from a simple runtime heap, and then storing the name of the thief worker and the name of the thief subcomputation (as contained in the steal request message) in the assignment information. Finally, the victim worker sends a copy of the closure to the thief. When the thief receives the stolen closure, it records the name of the victim worker in its thief subcomputation, and it places the closure in the subcomputation's ready pool. Now the thief subcomputation is ready, and the thief worker may commence executing it.

When a worker finishes executing a thief subcomputation, the link between the thief subcomputation and its victim closure is destroyed. Specifically, when a subcomputation has no closures in any of its three pools, then the subcomputation is *finished*. A worker with a finished thief subcomputation sends a message containing the subcomputation's name to the subcomputation's victim worker. Using this name, the victim worker finds the victim closure. This closure is removed from its subcomputation's assigned pool and then the closure and its assignment information are freed. The victim worker then acknowledges the message, and when the thief worker receives the acknowledgment, it frees the subcomputation. When the root subcomputation is finished, the entire Cilk job is finished.

In addition to allocating a new subcomputation, whenever a worker steals a closure, it alters any continuations in the closure so that they all refer to closures within the same subcomputation. Consider a thief stealing a closure, and suppose the victim closure contains a continuation referring to a closure that we call the *target*. The victim and target closures must be in the same subcomputation in the victim worker. Continuations are implemented as the address of the target closure concatenated with the index of an argument slot in the target closure. Therefore, the continuation in the victim closure contains the address of the target closure, and this address is only meaningful to the victim worker. When the thief worker receives the stolen closure, it replaces the continuation with a new continuation referring to a new "result" closure, as follows. First, the thief must locate the continuation with the aid of a *thread signature*. For each thread in the program, the `cilk2c` translator creates a signature which specifies the thread address (a pointer to the thread's code) and the type of each argument to the thread. All of the thread signatures are stored in a table. To find the continuations in a closure, the worker uses the closure's thread to lookup the signature in the table. Then the worker uses the signature to locate which arguments are continuations. Having located the continuation in the stolen closure, the thief allocates a new *result* closure and replaces the continuation with a new continuation referring to a slot in the result closure, as illustrated in Figure 6.5. The result closure's thread is a special system thread whose operation we shall explain shortly. This thread takes two arguments: a *continuation index* and a *result value*. The continuation index is supplied as the continuation's argument slot number in the stolen closure. The result value is missing, and the continuation in the stolen closure is set to refer to this argument slot. The result closure is waiting and its join counter is 1. In general, the thief allocates a result closure and performs this alteration for each continuation in the stolen closure. The stolen and result closures are part of the same subcomputation.

Using continuations to send values from one thread to another operates as described in Section 5.1, but when a value is sent to a result closure, communication between different subcomputations occurs. When a result closure receives its result value, it becomes

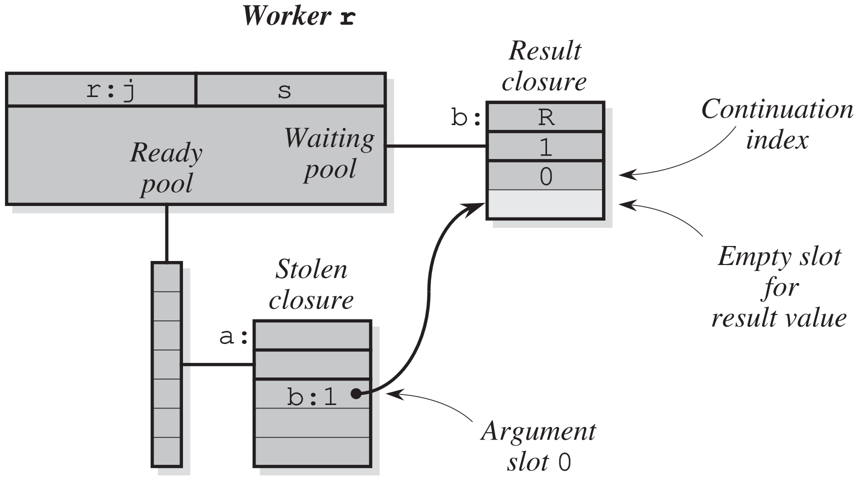



**Figure 6.5**: When the thief worker `r` steals a closure `a` which contains a continuation in its argument slot `0`, the thief replaces this continuation with a new continuation referring to an empty slot in a newly allocated result closure `b`. Additionally, the thief stores the continuation's slot number `0` in an argument slot of the result closure.

ready, and when its thread executes, it forwards the result value to another closure in another subcomputation as follows. When a worker executing a subcomputation executes a result closure's thread, it sends a message to the subcomputation's victim worker. This message contains the subcomputation's name as well as the continuation index and result value that are the thread's arguments. When the victim worker receives this message, it uses the subcomputation name to find the victim closure. Then it uses the continuation index to find a continuation in the victim closure. Finally, it uses this continuation to send the result value to the target.

To summarize, each subcomputation contains a collection of closures and every continuation in a closure refers to another closure in the same subcomputation. To send a value from a closure in one subcomputation to a closure in another, the value must be forwarded through intermediate result closures passing from subcomputation to subcomputation by way of the links between thief subcomputations and victim closures. All relations between different subcomputations are confined to these links.

With this structure, migrating a subcomputation from one worker `x` to another worker `y` is fairly straightforward. First, all of the subcomputation's closures are *swizzled* in order to replace addresses with values that have meaning independent of any particular worker's address space as follows. The thread field in each closure is translated from an address to an index in the table of thread signatures, and each closure is assigned a unique index. Then, using the thread signatures again, for each continuation in each closure, the pointer portion of the continuation is replaced with the target closure's index. Having swizzled all of its closures, the subcomputation and the closures are sent in messages from worker `x` to worker `y`. The subcomputation keeps its name. When the entire subcomputation and all of its closures have been migrated to worker `y`, the closures are *unswizzled* to replace the thread and closure indices with actual addresses. Finally, worker `y` sends a message to the subcomputation's victim worker to inform the victim closure of its thief subcomputation's new thief worker. Additionally, for each of the subcomputation's assigned closures, it sends a message to the thief worker to inform the thief subcomputation of its victim closure's new victim worker. Thus, all of the links between victim closures and thief subcomputations are preserved.

Adaptive parallelism proved extremely valuable in the protein-folding experiment described at the top of this chapter. Figure 6.6 plots the number of machines that were idle at each point in time over the course of a typical week for our network of 50 SparcStations.[1] As can be seen from this plot, though many more machines are idle at night, a significant number of machines are idle at various times throughout the day. Therefore, by adaptively using idle machines both day and night, we can take advantage of significantly more machine resources than if we run our parallel jobs as batch jobs during the night. During the day, however, a given machine is less likely to remain idle for an extended period of time. Thus, in order to efficiently use machines adaptively during the day, the runtime system must be able to utilize potentially short intervals of idle time.

To document the efficiency with which Cilk-NOW can utilize short periods of idle time, we ran the following experiment. We used 8 SparcStation 1+ workstations connected by Ethernet, and we ran the `knary(11,6,2)` program (see page 78) several times with varying degrees of adaptiveness. We controlled the amount of adaptiveness by killing each worker and then starting a new worker at fixed intervals on each machine. Figure 6.7 shows the results of this experiment. The horizontal position of each plotted datum is the amount of adaptiveness in the run as measured by the average execution time for each worker. For example, a datum plotted at a horizontal position of 50 seconds represents a run in which the average worker running on each of the 8 machines worked for 50 seconds before leaving the job. The `knary(11,6,2)` program performs approximately 2600 seconds of work, so such a run involves at least $2600/50 = 52$ total workers. The vertical position of each plotted datum is the efficiency of the run as measured by taking the ratio of the work in the computation (or execution time with one worker and no adaptiveness) to the sum of the execution times of each worker in the run. If the efficiency is 1.0, then the total worker execution time equals the work of the computation and every worker is utilized with perfect efficiency. This

[1] The node manager's idleness predicate on all 50 machines was conservatively set to require that the keyboard and mouse have not been touched for 15 minutes and the 1, 5, and 15 minute processor load averages are below 0.35, 0.30, and 0.25 respectively.

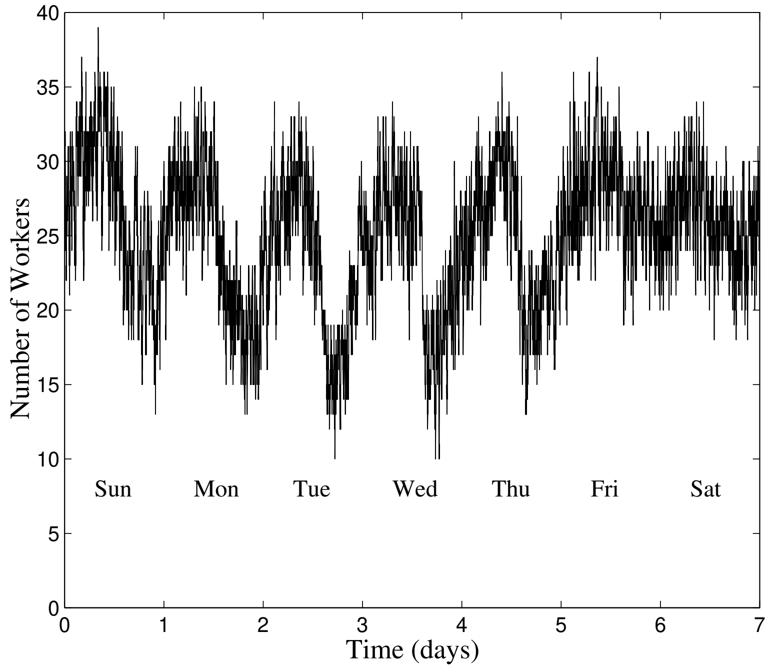



**Figure 6.6**: This plot shows the number of machines, out of the 50 machines in our network, that are idle over the course of one typical week in March, 1995.

plot shows that even when the average worker stays with the job for only 5 seconds, the efficiency is still over 85 percent, and if the average worker stays with the job for 1 minute, then the efficiency is generally over 95 percent.

## 6.3 Fault tolerance

With transparent fault tolerance built into the Cilk-NOW runtime system, Cilk jobs may survive machine crashes or network outages despite the fact that Cilk programs are fault oblivious, having been coded with no special provision for handling machine or network failures. If a worker crashes, then other workers automatically redo any work that was lost in the crash. In the case of a more catastrophic failure, such as a power outage, a total network failure, or a crash of the file server, then all workers may crash. For this case, Cilk-NOW provides automatic checkpointing, so when service is restored, the Cilk job may be restarted with minimal lost work. In this section, we show how the structure used to support adaptive parallelism—which leverages strictness and the work-stealing scheduler—may be further leveraged to build these fault tolerant capabilities in Cilk-NOW.

Given adaptive parallelism, fault tolerance is only a short step away. With adaptive par-

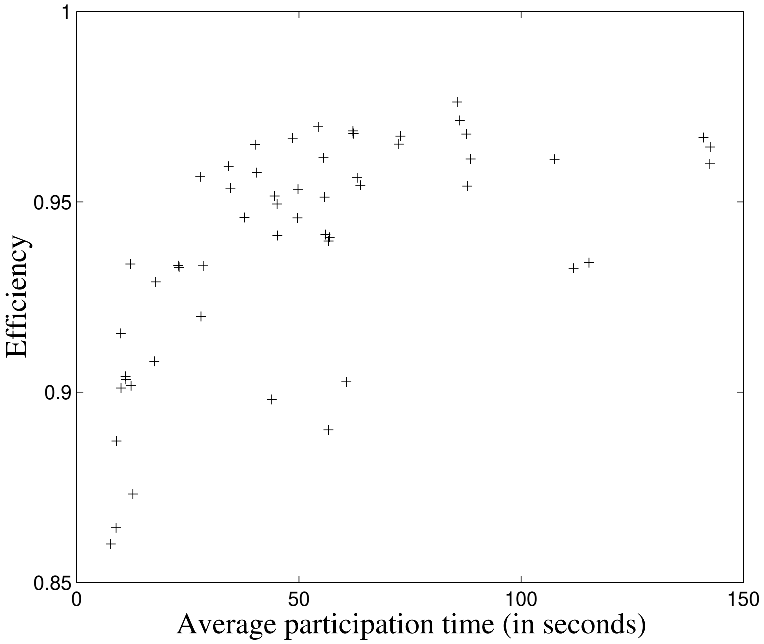



**Figure 6.7**: This plot shows the efficiency of 8-processor executions of the `knary(11,6,2)` program as a function of the amount of adaptiveness. The adaptiveness is measured as the average amount of time each worker participated in the job. The efficiency is measured as the ratio of the work in the computation to the total worker execution time.

allelism, a worker may leave a Cilk job, but before doing so, it first migrates all of its subcomputations to other workers. In contrast, when a worker crashes, all of its subcomputations are lost. To support fault tolerance, we add a mechanism that allows surviving workers to redo any work that was done by the lost subcomputations. Such a mechanism must address two fundamental issues. First, not all work is necessarily idempotent, so redoing work may present problems. We address this issue by ensuring that the work done by any given subcomputation does not affect the state of any other subcomputations until the given subcomputation finishes. Thus, from the point-of-view of any other subcomputation, the work of a subcomputation appears as a transaction: either the subcomputation finishes and commits its work by making it visible to other subcomputations, or the subcomputation never happened. Second, the lost subcomputations may have done a large amount of work, and we would like to minimize the amount of work that needs to be redone. We address this issue by incorporating a transparent and fully distributed checkpointing facility. This checkpointing facility also allows a Cilk job to be restarted in the case of a total system failure in which every worker crashes.

To make the work of a subcomputation appear from the outside as a transaction, we modify the behavior of the subcomputation's result closures by delaying their execution until the subcomputation finishes. Rather than set the join counter in each result closure to 1, we set the join counter to 2, so the result closure will never be ready and always wait. Additionally, rather than keep the result closures in the subcomputation's waiting pool, we keep them in a special *result pool*. When the subcomputation's ready, waiting, and assigned pools are all empty, then the subcomputation is finished, and the result closures may be executed. The thread executed by each of these closures sends a message to the subcomputation's victim worker. Also, the fact that the subcomputation is finished warrants a message to the victim worker. We bundle all of these messages into a single larger message sent to the victim worker. When the victim worker receives this message, it commits all of the thief subcomputation's work by sending the appropriate result values from the victim closure, freeing the victim closure, and sending an acknowledgment back to the thief worker.

By delaying the execution of the result closures, we have a very simple technique for making the work of a subcomputation appear as a transaction, but on the negative side, this delaying technique may result in a longer critical path. So far, none of our applications suffer from this effect, because all of our applications are fully strict and each procedure sends a value to its parent only as the last thing it does. Should the need arise, we may allow subcomputations to perform commits before they finish by tying these commits to checkpoints.

With subcomputations having this transactional nature, a Cilk job can tolerate individual worker crashes as follows. Suppose a worker crashes. Eventually, the clearinghouse will detect the crash, and the other living workers will learn of the crash at the next update from the clearinghouse. When a worker learns of a crash, it goes through all of its subcomputations, checking each assigned closure to see if it is assigned to the crashed worker. Each such closure is moved from the assigned pool back to the ready pool (and its assignment information is freed). Thus, all of the work done by the closure's thief subcomputation which has been lost in the crash will eventually be redone. Additionally, when a worker learns of a crash, it goes through all of its subcomputations to see if it has any that record the crashed worker as the subcomputation's victim. For each such subcomputation, the worker aborts it as follows. The worker goes through all of the subcomputation's assigned closures sending to each thief worker an *abort* message specifying the name of the thief subcomputation. Then the worker frees the subcomputation and all of its closures. When a worker receives an abort message, it finds the thief subcomputation named in the message and recursively aborts it. All of the work done by these aborted subcomputations must eventually be redone. In order to avoid aborting all of these subcomputations (which may comprise the entire job in the case when the root subcomputation is lost) and redoing potentially vast amounts of work, and in order to allow restarting when the entire job is lost, we need checkpointing.

Cilk-NOW performs automatic checkpointing without any synchronization among different workers and without any notion of global state. Specifically, each subcomputation is periodically checkpointed to a file named with the subcomputation's name. For example, a subcomputation named `r:i` would be checkpointed to a file named `scomp_r_i`. We assume that all workers in the job have access to a common file system (through NFS or AFS, for example), and all checkpoint files are written to a common checkpoint directory.[2] To write a checkpoint file for a subcomputation `r:i`, the worker first opens a file named `scomp_r_i.temp`. Then it swizzles all of the subcomputation's closures, as described in the previous section; writes the subcomputation record and all of the closures—including the assignment information for the assigned closures—into the file; and unswizzles the closures. Finally, it atomically renames the file `scomp_r_i.temp` to `scomp_r_i`, overwriting any previous checkpoint file. A checkpoint file can be read to recover the subcomputation by simply unswizzling the closures, as described in the previous section. On writing a checkpoint file, the worker additionally prunes any no-longer-needed checkpoint files as follows. Suppose that when the previous checkpoint file was written, the subcomputation had a closure assigned to a thief subcomputation `s:j`, but since then, the thief subcomputation has finished and the assigned closure has been freed. In this case, as soon as the new checkpoint file `scomp_r_i` is written, the checkpoint file `scomp_s_j` is no longer needed. The worker deletes the checkpoint file `scomp_s_j` as follows. It first reads the file, and for each of the subcomputation's assigned closures, it recursively deletes the thief subcomputation's checkpoint file, as this checkpoint file is also no longer needed. Finally, the worker actually removes the file.

If workers crash, the lost subcomputations can be recovered from checkpoint files. In the case of a single worker crash, the lost subcomputations can be recovered automatically. When a surviving worker finds that it has a subcomputation with a closure assigned to the crashed worker, then it can recover the thief subcomputation by reading the checkpoint file. It is possible that the checkpoint file may not exist, but in this case, the thief subcomputation, having not written a checkpoint file, cannot have done very much work, so little is lost by redoing the work as previously described. In the case of a large-scale failure in which every worker crashes, the Cilk job can be restarted from checkpoint files by setting the `-Recover` flag on the command line. Recovery begins with the root subcomputation whose checkpoint file is `scomp_0_1`. After recovering the root subcomputation, then every other subcomputation can be recovered by recursively recovering the thief subcomputation for each of the root subcomputation's assigned closures.

Without checkpointing, we could not have completed the protein-folding experiment

[2] We have not yet implemented any sort of distributed file system. In the current implementation, workers implicitly synchronize when they write checkpoint files, since they all access a common file system.

described at the top of this chapter. In addition to several single-machine crashes (mostly due to power cycling), we experienced a total system failure when our network's NFS server was down for maintenance. When the server came back on line, we were able to restart the program from where it left off.

# Chapter 7

# Conclusions

Though this thesis has established an algorithmic foundation on which to build parallel multithreaded systems, much work remains to be done. We have shown that strictness is a sufficient condition for efficient scheduling, though surely it is not a necessary condition. If we can identify useful forms of nonstrictness and develop provably efficient scheduling algorithms for these cases, then we can generalize the applicability of Cilk. Some forms of nonstrictness, for example, might allow Cilk to efficiently execute more synchronous types of applications.

Our current work is focused on incorporating in Cilk a distributed shared memory using *dag consistency* [11]. A distributed shared memory provides a global virtual address space, so an instruction may load from or store to a virtual address that has meaning independent of which processor executes the instruction. With *deterministic* dag consistency, we specify the value returned when an instruction performs a load as follows. All of the stores to any given virtual address must be totally ordered with respect to each other in the computation's dag of instructions. Additionally, each individual load from a virtual address must be totally ordered with respect to all of the stores to that address. Then, the value returned by a load instruction is the value stored by the immediately preceding store instruction in this total order. Dag consistency can be generalized to the case when no such total order exists by introducing nondeterministic execution. Dag consistency builds upon our algorithmic foundations, and for the case of fully strict computations, we can prove bounds on the number of page (or object) faults. We have recently implemented dag consistency and a simple "cactus-stack" memory allocator on the CM5, and we have coded several divide-and-conquer applications including blocked matrix multiply, Strassen's matrix multiply, and a Barnes-Hut $N$-body simulation. Cilk together with dag consistency makes coding these applications particularly simple. Strassen's algorithm was coded in one evening. Preliminary results with these applications is extremely encouraging, though we do not have numbers to report here. We further plan to add dag consistent shared memory to Cilk-NOW. We have proven that dag consistency can be maintained using a simple and efficient checkout/commit algorithm, and the nature of this algorithm dovetails perfectly into Cilk-NOW's implementation of adaptive parallelism and fault tolerance.

This thesis has shown that by building a multithreaded language and runtime system on top of an algorithmic foundation, programmers can focus on expressing the parallelism in their algorithm and leave the runtime system's scheduler to manage the low-level details of scheduling, secure and confident in the scheduler's ability to deliver performance. Cilk is just such a language and runtime system. In Cilk we can delineate a specific class of programs—fully strict programs—and for this class, we can guarantee efficient execution and performance that is predictable with a simple model based on work and critical path length. Thus, besides the programming abstractions provided by the language, Cilk provides a performance abstraction. We argue that rather than think about machine-specific quantities such as execution time and communication costs, the performance-minded programmer should instead think about program abstractions such as work and critical path length and then rely on the runtime system's performance model to translate these abstract quantities into real quantities. The value of Cilk's performance abstraction is intimately linked to its provable efficiency. Abstraction requires guarantees.

# Appendix A

# Proof of lower bound

**Theorem 3.1** *For any $S_1 \geq 4$ and any $T_1 \geq 16S_1^2$, there exists a depth-first multithreaded computation with work $T_1$, average parallelism $T_1/T_\infty \geq \sqrt{T_1}/8$, and stack depth $S_1$ such that the following holds. For any number $P$ of processors and any value $\rho$ in the range $1 \leq \rho \leq \frac{1}{8}T_1/T_\infty$, if $\mathcal{X}$ is a P-processor execution schedule that achieves speedup $\rho$—that is, $T(\mathcal{X}) \leq T_1/\rho$—then $S(\mathcal{X}) \geq \frac{1}{4}(\rho - 1)\sqrt{T_1} + S_1$.*

*Proof:* To exhibit the desired depth-first multithreaded computation with work $T_1$, critical path length $T_\infty$, and stack depth $S_1$, we first ignore the partitioning of instructions into threads and consider just the dag structure of the computation. Minus a few instructions and dependencies, the dag appears as in Figure A.1(a). The instructions are organized into

$$m = \sqrt{T_1}/8$$

separate components $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{m-1}$ that we call *chains*.[1] Each chain begins with

$$\lambda = \sqrt{T_1}/S_1$$

instructions that we call *headers* (vertical hashed in Figure A.1(a)). After the headers, each chain contains

$$\nu = 6\sqrt{T_1}$$

instructions (plain white in Figure A.1(a)) that form the *trunk*. At the end of each chain, there are $\lambda$ *blockers* (horizontal hashed in Figure A.1(a)). Each chain, therefore, consists of $2\lambda + \nu = 2(\sqrt{T_1}/S_1) + 6\sqrt{T_1}$ instructions. Since there are $m = \sqrt{T_1}/8$ chains, the total

[1] In what follows, we refer to a number $x$ of objects (such as instructions) when $x$ may not be integral. Rounding these quantities to integers does not affect the correctness of the proof. For ease of exposition, we shall not consider the issue.

number of instructions accounted for by the $m$ chains is $(2\sqrt{T_1}/S_1+6\sqrt{T_1})\sqrt{T_1}/8=\frac{3}{4}T_1+\frac{1}{4}T_1/S_1$, and this number is no more than $\frac{13}{16}T_1$ since $S_1\geq 4$. The remaining (at least) $\frac{3}{16}T_1$ instructions form the parts of the computation not shown in Figure A.1(a).

There are no dependencies between different chains so the average parallelism $T_1/T_\infty$ is at least $m=\sqrt{T_1}/8$ and the critical path length $T_\infty$ is no more than $8\sqrt{T_1}$ as promised.

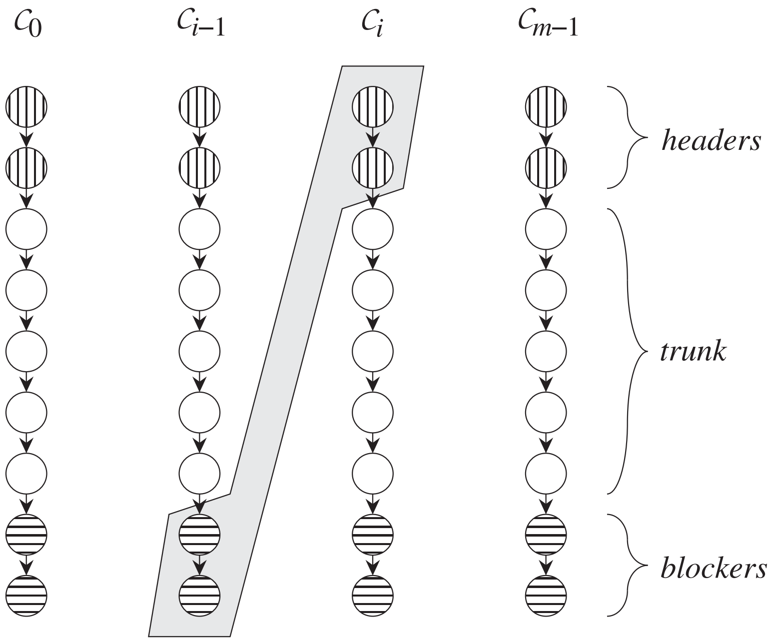
Now consider the partitioning of the instructions from each chain into the actual threads. As alluded to in Figure A.1(b), the root thread has $m-1$ child threads, each of which is the root of a subcomputation that we call an *outer iteration*. (The outer iterations contain *inner iterations* that will be discussed later.) Each of these outer iterations contains $\sqrt{T_1}/2$ threads. As indicated by the shading in Figure A.1, the $i$th outer iteration for $i=1,2,\ldots,m-1$ contains both the header instructions of chain $C_i$ and the blocker instructions of chain $C_{i-1}$. These instructions are organized into the threads of the outer iteration so as to ensure that chain $C_i$ cannot begin executing its trunk instructions until all $\sqrt{T_1}/2$ of the outer iteration's threads have been spawned, and none of these threads can die until chain $C_{i-1}$ begins executing its blocker instructions. (We will exhibit this organization later.) Thus, if chain $C_i$ begins executing its trunk instructions before chain $C_{i-1}$ finishes its, then the execution will require at least $\sqrt{T_1}/2$ space.

For any number $P$ of processors, consider any valid $P$-processor execution schedule $\mathcal{X}$. For each chain $C_i$, let $t_i^{(\mathrm{s})}$ denote the time step at which $\mathcal{X}$ executes the first trunk instruction of $C_i$, and let $t_i^{(\mathrm{f})}$ denote the first time step at which $\mathcal{X}$ executes a blocker instruction of $C_i$. Since the trunk has length $\nu$ and no blocker instruction of $C_i$ can execute until after the last trunk instruction of $C_i$, we have $t_i^{(\mathrm{f})}-t_i^{(\mathrm{s})}\geq\nu$.

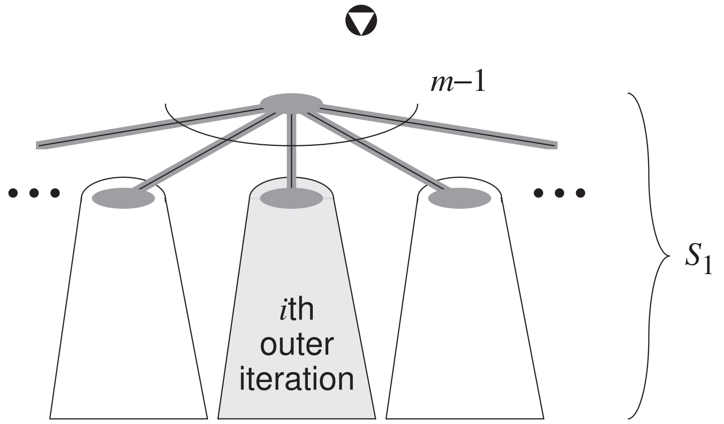
Now consider two chains, $C_i$ and $C_{i-1}$, and suppose $t_i^{(\mathrm{s})}<t_{i-1}^{(\mathrm{f})}$; this is the scenario we described as using at least $\sqrt{T_1}/2$ space. In this case, we consider the time interval from $t_i^{(\mathrm{s})}$ (inclusive) to $t_{i-1}^{(\mathrm{f})}$ (exclusive) during which we say that chain $C_i$ is *exposed*, and we let $\tau_i=t_{i-1}^{(\mathrm{f})}-t_i^{(\mathrm{s})}$ denote the amount of time chain $C_i$ is exposed. See Figure A.2. If $t_i^{(\mathrm{s})}\geq t_{i-1}^{(\mathrm{f})}$ then chain $C_i$ is never exposed and we let $\tau_i=0$. As we have seen, over the time interval during which a chain is exposed, it uses at least $\sqrt{T_1}/2$ space. We will show that in order for an execution schedule $\mathcal{X}$ to achieve speedup $\rho$—that is $T(\mathcal{X})\leq T_1/\rho$—there must be some time step during the execution at which at least $\lceil\frac{3}{4}\rho\rceil-1$ chains are exposed.

If schedule $\mathcal{X}$ is such that $T(\mathcal{X})\leq T_1/\rho$, then we must have $t_{m-1}^{(\mathrm{f})}-t_0^{(\mathrm{s})}\leq T_1/\rho$. We can expand this inequality to yield

$$\begin{aligned} T_1/\rho &\geq t_{m-1}^{(\mathrm{f})}-t_0^{(\mathrm{s})} \\ &= \sum_{i=0}^{m-1}\left(t_i^{(\mathrm{f})}-t_i^{(\mathrm{s})}\right)-\sum_{i=1}^{m-1}\left(t_{i-1}^{(\mathrm{f})}-t_i^{(\mathrm{s})}\right). \end{aligned} \tag{A.1}$$




**(a)** Chains of instructions




**(b)** Outer iterations

**Figure A.1**: Constructing a computation with no efficient execution schedule. The header instructions of chain $\mathcal{C}_i$ and the blocker instructions of chain $\mathcal{C}_{i-1}$ are both placed in the threads of the $i$th outer iteration.

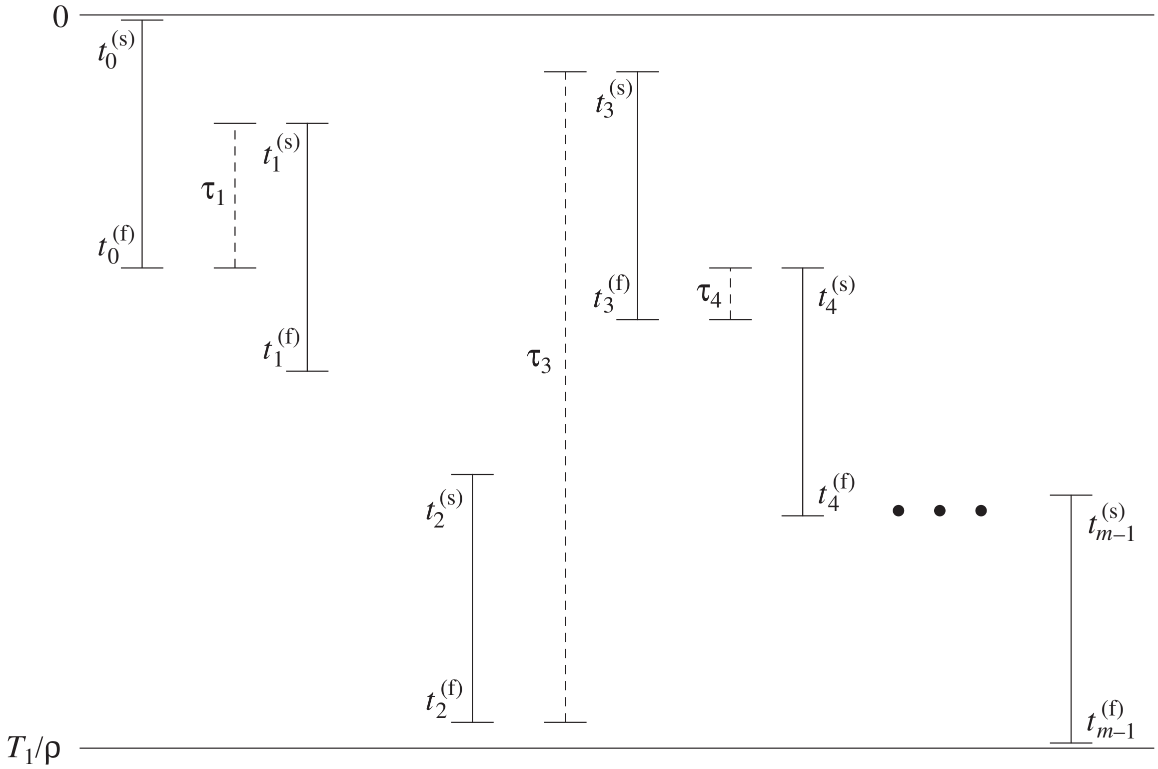



**Figure A.2**: Scheduling the execution of the chains. A solid vertical interval from $t_i^{(s)}$ to $t_i^{(f)}$ indicates the time during which the trunk of chain $C_i$ is being executed. When $t_i^{(s)} < t_{i-1}^{(f)}$, we can define an interval, shown dashed, of length $\tau_i = t_{i-1}^{(f)} - t_i^{(s)}$, during which chain $C_i$ is exposed.

Considering the first sum, we recall that $t_i^{(f)} - t_i^{(s)} \geq \nu$, hence,

$$\sum_{i=0}^{m-1} (t_i^{(f)} - t_i^{(s)}) \geq m\nu \; . \tag{A.2}$$

Considering the second sum of Inequality (A.1), when $t_{i-1}^{(f)} > t_i^{(s)}$ (so $C_i$ is exposed), we have $\tau_i = t_{i-1}^{(f)} - t_i^{(s)}$, and otherwise, $\tau_i = 0 \geq t_{i-1}^{(f)} - t_i^{(s)}$. Therefore,

$$\sum_{i=1}^{m-1} (t_{i-1}^{(f)} - t_i^{(s)}) \leq \sum_{i=1}^{m-1} \tau_i \; . \tag{A.3}$$

Substituting Inequality (A.2) and Inequality (A.3) back into Inequality (A.1), we obtain

$$\sum_{i=1}^{m-1} \tau_i \geq m\nu - T_1/\rho \; .$$

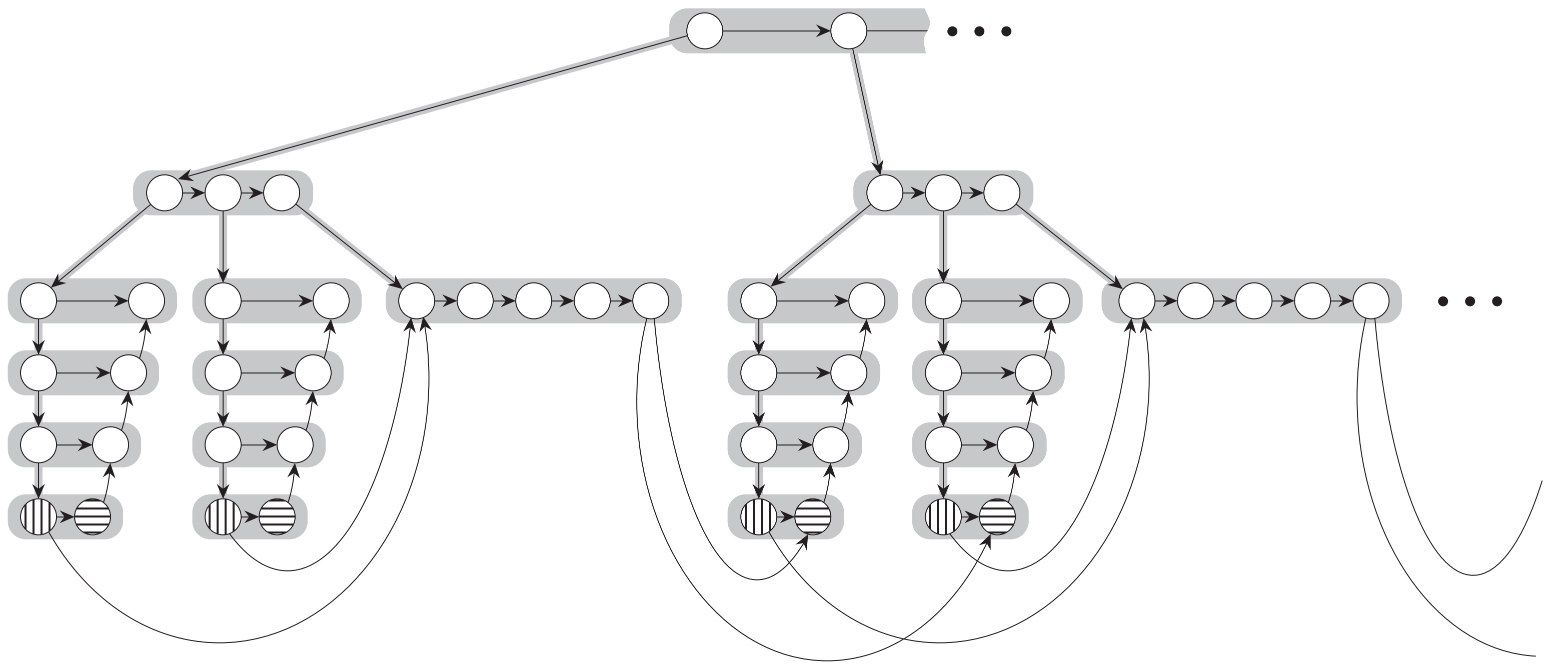


**Figure A.3**: Laying out the chains into the threads of a multithreaded computation. As before, the header instructions are vertical hashed, and the blocker instructions are horizontal hashed. In this example, each activation frame has unit size so $\mathcal{S}=6$. Also, in this example $\lambda=2$, $\nu=5$, and only the first 2 out of the $m$ instructions in the root thread are shown. Each instruction of the root thread spawns a child (an outer iteration), and each child thread contains $\lambda+1=3$ instructions; the first $\lambda$ of these spawn a child thread which is the root of an inner iteration with stack depth $\mathcal{S}-2=4$, and the last one spawns a leaf thread with the $\nu=5$ trunk instructions of a single chain.

Let $exposed(t)$ denote the number of chains exposed at time step $t$, and observe that

$$\sum_{t=1}^{T_1/\rho} exposed(t) = \sum_{i=i}^{m-1} \tau_i \ .$$

Then the average number of exposed chains per time step is

$$\begin{aligned} \frac{1}{T_1/\rho} \sum_{t=1}^{T_1/\rho} exposed(t) &= \frac{1}{T_1/\rho} \sum_{i=1}^{m-1} \tau_i \\ &\geq \frac{1}{T_1/\rho} (m\nu - T_1/\rho) \\ &= \frac{3}{4}\rho - 1 \end{aligned}$$

since, $m = \sqrt{T_1}/8$ and $\nu = 6\sqrt{T_1}$. There must be some time step $t^*$ for which $exposed(t^*)$ is at least the average, and consequently,

$$exposed(t^*) \geq \left\lceil \frac{3}{4}\rho \right\rceil - 1 \ .$$

Now, recalling that each exposed chain uses space $\sqrt{T_1}/2$, we have

$$\begin{aligned} S(\mathcal{X}) &\geq \left( \left\lceil \frac{3}{4}\rho \right\rceil - 1 \right) \frac{1}{2} \sqrt{T_1} \\ &\geq \frac{1}{4}(\rho - 1)\sqrt{T_1} + S_1 \end{aligned}$$

for $S_1 \leq \sqrt{T_1}/4$ (which is true since $T_1 \geq 16S_1^2$).

All that remains is exhibiting the organization of the instructions of each chain into a depth-first multithreaded computation with work $T_1$, critical path length $T_\infty \leq 8\sqrt{T_1}$, and stack depth $\mathcal{S} = S_1$ in such a way that each exposed chain uses $\sqrt{T_1}/2$ space. There are actually many ways of creating such a computation. One such way, that uses unit size activation frames for each thread, is shown in Figure A.3.

For the multithreaded computation of Figure A.3, the root thread contains $m$ instructions, each of which spawns a child thread (an outer iteration). Each child thread contains $\lambda + 1$ instructions; the first $\lambda$ of these spawn a child thread which is the root of a subcomputation that we call an *inner iteration*. Each inner iteration has stack depth $S_1 - 2 \geq S_1/2$ (since $S_1 \geq 4$), and the last one spawns a leaf thread with the $\nu$ trunk instructions of a single chain. Each of these inner iterations contains a single header from one chain and a single blocker from the previous chain (except in the case of the first group of $\lambda$) as shown in Figure A.3.

The header and blocker in an inner iteration are organized such that in order to execute the header, all $S_1 - 2$ of the threads in the inner iteration must be spawned, and none of them can die until the blocker executes. Thus, when a chain is exposed, all $\lambda$ of these inner iterations have all of their threads living, thereby using space $\lambda(S_1 - 2) \geq (\sqrt{T_1}/S_1)(S_1/2) = \sqrt{T_1}/2$.

We can verify from Figure A.3 and from the given values of $m$, $\lambda$, and $\nu$ that this construction actually has work slightly less than $T_1$; in order to make the work equal to $T_1$ we can just add the extra instructions evenly among the threads that contain the trunk of each chain (thereby increasing $\nu$ by a bit). Also, we can verify that $T_\infty \leq 8\sqrt{T_1}$. Finally, looking at Figure A.3 we can see that this computation is indeed depth-first. ■

The construction of a multithreaded computation with provably bad time/space characteristics as just described can be modified in various ways to accommodate various restrictions to the model while still obtaining the same result. For example, some real multithreaded systems require limits on the number of instructions in a thread, dependencies that only go to the first instruction of a thread, limited fan-in for dependencies, or a limit on the number of children a thread can have. Simple changes to the construction just described can produce multithreaded computations that accommodate any or all of these restrictions and still have the same provably bad time/space tradeoff. Thus, the lower bound of Theorem 3.1 holds even for multithreaded computations with any or all of these restrictions.

# Appendix B

# Cilk project status

Up to date Cilk information, papers, and software releases are available on the World Wide Web (WWW) at the following two locations:

☞ `http://theory.lcs.mit.edu/~cilk`

☞ `http://www.cs.utexas.edu/users/cilk`

The Cilk team can be contacted by electronic mail at the following location:

✉ `cilk-developers@theory.lcs.mit.edu`

Contact the team to get on the `cilk-users` mailing list.

The current software release is version 2.0. This release is essentially as described in Chapter 5. The Cilk 2.0 release includes the Cilk-to-C translator `cilk2c`, a collection of example programs, a reference manual, and runtime system support for two multiprocessor platforms—the Thinking Machines CM5 and the Sun SparcStation SMP—and uniprocessor platforms running SunOS or Linux. Cilk has also been ported to PVM, the Intel Paragon MPP, and the Silicon Graphics Power Challenge SMP, but these ports are not included in the current release. For ports to other machines, the *Cilk Reference Manual* includes an *Implementor's Guide*. Contact the Cilk team for help porting Cilk to a new platform.

The next major release, Cilk 3.0, will include distributed shared memory.

A release of the Cilk-NOW runtime system is planned for the near future. This release will support Cilk 2.0. A release supporting Cilk 3.0 will come sometime later. Watch the web pages or get on the `cilk-users` mailing list to keep abreast of forthcoming software releases.

*August, 1995*

# Bibliography


[1] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. John Wiley & Sons, 1992.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[3] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[4] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Digest of Papers from the Thirty-Eighth IEEE Computer Society International Conference (Spring COMPCON)*, pages 528–537, San Francisco, California, February 1993.

[5] Sandeep Bhatt, David Greenberg, Tom Leighton, and Pangfeng Liu. Tight bounds for on-line tree embeddings. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 344–350, San Francisco, California, January 1991.

[6] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP 14)*, pages 217–230, Asheville, North Carolina, December 1993.

[7] Andrew D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Equipment Corporation Systems Research Center, January 1989.

[8] Guy E. Blelloch. Programming parallel algorithms. In *Proceedings of the 1992 Dartmouth Institute for Advanced Graduate Studies (DAGS) Symposium on Parallel Computation*, pages 11–18, Hanover, New Hampshire, June 1992.

[9] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, California, July 1995.

[10] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Rob Miller, Keith H. Randall, and Yuli Zhou. *Cilk 2.0 Reference Manual*. MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139, June 1995.

[11] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, April 1996. to appear.

[12] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[13] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 362–371, San Diego, California, May 1993.

[14] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[15] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing (HPDC)*, pages 96–105, San Francisco, California, August 1994.

[16] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[17] Eric A. Brewer and Robert Blumofe. Strata: A multi-layer communications library. Technical Report to appear, MIT Laboratory for Computer Science. Available as `ftp://ftp.lcs.mit.edu/pub/supertech/strata/strata.tar.Z`.

[18] F. Warren Burton. Storage management in virtual tree machines. *IEEE Transactions on Computers*, 37(3):321–328, March 1988.

[19] F. Warren Burton and David J. Simpson. Space efficient execution of deterministic parallel programs. Unpublished manuscript, 1994.

[20] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, October 1981.

[21] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

[22] Martin C. Carlisle, Anne Rogers, John H. Reppy, and Laurie J. Hendren. Early experiences with Olden. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[23] Nicholas Carriero, Eric Freeman, and David Gelernter. Adaptive parallelism on multiprocessors: Preliminary experience with Piranha on the CM-5. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[24] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[25] Vint Cerf and Robert Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Computers*, 22(5):637–648, May 1974.

[26] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, San Jose, California, October 1994.

[27] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 249–259, San Diego, California, May 1993.

[28] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[29] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP 12)*, pages 147–158, Litchfield Park, Arizona, December 1989.

[30] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[31] Andrew A. Chien and William J. Dally. Concurrent Aggregates (CA). In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 187–196, Seattle, Washington, March 1990. Also: MIT Artificial Intelligence Laboratory Technical Report MIT/AI/TR-1248.

[32] Henry Clark and Bruce McMillin. DAWGS—a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14(2):175–186, February 1992.

[33] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[34] David E. Culler. Resource management for the tagged token dataflow architecture. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1980. Also: MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-332.

[35] David E. Culler. *Managing Parallelism and Resources in Scientific Dataflow Programs*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, March 1990. Also: MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-446.

[36] David E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150, Honolulu, Hawaii, May 1988. Also: MIT Laboratory for Computer Science, Computation Structures Group Memo 280.

[37] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 1991.

[38] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a message-driven processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–196, Pittsburgh, Pennsylvania, June 1987. Also: MIT Artificial Intelligence Laboratory Technical Report MIT/AI/TR-1069.

[39] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill, James R. Larus, Anne Rogers, and David A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing '94*, pages 380–389, Washington, D.C., November 1994.

[40] Robert E. Felderman, Eve M. Schooler, and Leonard Kleinrock. The Benevolent Bandit Laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 7(2):303–311, February 1989.

[41] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. Studying overheads in massively parallel min/max-tree evaluation. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–103, Cape May, New Jersey, June 1994.

[42] Raphael Finkel and Udi Manber. DIB—a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[43] The MPI Forum. MPI: A message passing interface. In *Supercomputing '93*, pages 878–883, Portland, Oregon, November 1993.

[44] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

[45] Matteo Frigo, June 1995. Private communication.

[46] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 417–427, Washington, D.C., July 1992.

[47] Seth Copen Goldstein, Klaus Erik Schauser, and David Culler. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Troy, New York, May 1995.

[48] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45:1563–1581, November 1966.

[49] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[50] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994.

[51] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 1984.

[52] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[53] W. Hillis and G. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[54] Waldemar Horwat, Andrew A. Chien, and William J. Dally. Experience with CST: Programming and implementation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, Portland, Oregon, June 1989.

[55] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 239–248, San Diego, California, May 1993.

[56] Suresh Jagannathan and Jim Philbin. A customizable substrate for concurrent languages. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 55–67, San Francisco, California, June 1992.

[57] Suresh Jagannathan and Jim Philbin. A foundation for an efficient multi-threaded Scheme system. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 345–357, San Francisco, California, June 1992.

[58] Chris Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge*, Rutgers University, New Jersey, October 1994. Available as `ftp://theory.lcs.mit.edu/pub/cilk/dimacs94.ps.Z`.

[59] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 213–228, Copper Mountain Resort, Colorado, December 1995.

[60] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6:109–133, February 1988.

[61] Christos Kaklamanis and Giuseppe Persiano. Branch-and-bound and backtrack search on mesh-connected arrays of processors. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 118–126, San Diego, California, June 1992.

[62] David Louis Kaminsky. *Adaptive Parallelism with Piranha*. PhD thesis, Yale University, May 1994.

[63] Vijay Karamcheti and Andrew Chien. Concert—efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Supercomputing '93*, pages 598–607, Portland, Oregon, November 1993.

[64] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science—Volume A: Algorithms and Complexity*, chapter 17, pages 869–941. MIT Press, Cambridge, Massachusetts, 1990.

[65] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[66] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter 1994 Conference Proceedings*, pages 115–132, San Francisco, California, January 1994.

[67] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, Portland, Oregon, June 1989.

[68] Phillip Krueger and Rohit Chawla. The Stealth distributed scheduler. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 336–343, Arlington, Texas, May 1991.

[69] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford Flash multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, April 1994.

[70] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-645 or `ftp://theory.lcs.mit.edu/pub/bradley/phd.ps.Z`.

[71] Tom Leighton, Mark Newman, Abhiram G. Ranade, and Eric Schwabe. Dynamic tree embeddings in butterflies and hypercubes. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 224–234, Santa Fe, New Mexico, June 1989.

[72] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, San Diego, California, June 1992.

[73] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[74] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor—a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, California, June 1988.

[75] Pangfeng Liu, William Aiello, and Sandeep Bhatt. An atomic model for message-passing. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 154–163, Velen, Germany, June 1993.

[76] Robert C. Miller. A type-checking preprocessor for Cilk 2, a multithreaded C language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.

[77] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[78] B. Clifford Neuman and Santosh Rao. The Prospero resource manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 6(4):339–355, June 1994.

[79] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP 11)*, pages 5–12, Austin, Texas, November 1987.

[80] Rishiyur S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 390–405, Portland, Oregon, August 1993. Springer-Verlag.

[81] Rishiyur S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[82] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[83] Vijay S. Pande, Christopher F. Joerg, Alexander Yu Grosberg, and Toyoichi Tanaka. Enumerations of the hamiltonian walks on a cubic sublattice. *Journal of Physics A*, 27, 1994.

[84] Joseph D. Pehoushek and Joseph S. Weening. Low-cost process creation and dynamic partitioning in Qlisp. In *Parallel Lisp: Languages and Systems. Proceedings of the US/Japan Workshop*, number 441 in Lecture Notes in Computer Science, pages 182–199, Sendai, Japan, June 1989. Springer-Verlag.

[85] C. Gregory Plaxton, August 1994. Private communication.

[86] Abhiram Ranade. Optimal speedup for backtrack search on a butterfly network. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 40–48, Hilton Head, South Carolina, July 1991.

[87] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Chicago, Illinois, April 1994.

[88] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993.

[89] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, South Carolina, July 1991.

[90] Carlos A. Ruggiero and John Sargeant. Control of parallelism in the Manchester dataflow machine. In *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, 1987.

[91] Vikram A. Saletore, J. Jacob, and M. Padala. Parallel computations on the CHARM heterogeneous workstation cluster. In *Proceedings of the Third International Symposium on High Performance Distributed Computing*, pages 203–210, San Francisco, California, August 1994.

[92] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 101–114, Monterey, California, November 1994.

[93] Daniel J. Scales and Monica S. Lam. An efficient shared memory system for distributed memory machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.

[94] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, July 1994.

[95] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[96] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[97] V. S. Sunderam and Vernon J. Rego. EcliPSe: A system for high performance concurrent simulation. *Software—Practice and Experience*, 21(11):1189–1219, November 1991.

[98] Andrew S. Tanenbaum, Henri E. Bal, and M. Frans Kaashoek. Programming a distributed system using shared objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pages 5–12, Spokane, Washington, July 1993.

[99] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[100] Kenneth R. Traub. *Implementation of Non-Strict Functional Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1991.

[101] Andrew Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. PhD thesis, Stanford University, December 1993.

[102] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP 12)*, pages 159–166, Litchfield Park, Arizona, December 1989.

[103] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[104] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-latency communication over ATM networks using active messages. *IEEE Micro*, 15(1):46–53, February 1995.

[105] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[106] I-Chen Wu. Efficient parallel divide-and-conquer for a class of interconnection topologies. In *Proceedings of the 2nd International Symposium on Algorithms*, number 557 in Lecture Notes in Computer Science, pages 229–240, Taipei, Republic of China, December 1991. Springer-Verlag.

[107] I-Chen Wu and H. T. Kung. Communication complexity for parallel divide-and-conquer. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 151–162, San Juan, Puerto Rico, October 1991.

[108] Y. Zhang and A. Ortynski. The efficiency of randomized parallel backtrack search. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, October 1994. To appear.

[109] Yanjun Zhang. *Parallel Algorithms for Combinatorial Search Problems*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, November 1989. Also: University of California at Berkeley, Computer Science Division, Technical Report UCB/CSD 89/543.

[110] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. *Software—Practice and Experience*, 23(12):1305–1336, December 1993.