Managing Storage for Multithreaded Computations

by

Robert D. Blumofe

Sc.B., Brown University (1988)

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1992

Signature of Author . . . .
Department of Electrical Engineering and Computer Science
September 8, 1992

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

Accepted by . . . .
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Managing Storage for Multithreaded Computations

by

Robert D. Blumofe

Submitted to the Department of Electrical Engineering and Computer Science
on September 8, 1992, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Multithreading has become a dominant paradigm in general purpose MIMD parallel computation. To execute a multithreaded computation on a parallel computer, a scheduler must order and allocate threads to run on the individual processors. The scheduling algorithm dramatically affects both the speedup attained and the space used when executing the computation. We consider the problem of scheduling multithreaded computations to achieve linear speedup without using significantly more space-per-processor than required for a single-processor execution.

We show that for general multithreaded computations, no scheduling algorithm can simultaneously make efficient use of space and time. In particular, we show that there exist multithreaded computations such that any execution schedule $\mathcal{X}$ that achieves P-processor execution time $T_P(\mathcal{X}) \le T_1/\rho$, where T_1 is the minimum possible serial execution time, must use space at least $S_P(\mathcal{X}) \ge \frac{1}{4}(\rho - 1)\sqrt{T_1} + S_1$, where S_1 is the space used by an efficient serial execution. For such a computation, even achieving a factor of 2 speedup ($\rho = 2$) requires space proportional to the square root of the serial execution time.

By restricting ourselves to a class of computations we call *strict* computations, however, we show that there exist schedulers that can provide both efficient speedup and use of space. Specifically, we show that for any strict multithreaded computation and any number P of processors, there exists an execution schedule $\mathcal{X}$ that achieves time $T_P(\mathcal{X}) \le T_1/P + T_\infty$, where T_∞ is a lower bound on execution time even for arbitrarily large numbers of processors, and space $S_P(\mathcal{X}) \le S_1 P$. We demonstrate such schedules by exhibiting a simple centralized algorithm to compute them. We give a second, somewhat more efficient, algorithm that computes equally good execution schedules; this algorithm is online and should be practical for moderate numbers of processors, but its use of a centralized queue makes it inefficient for large numbers of processors.

To demonstrate an algorithm that is efficient even for large machines, we give a randomized, distributed, and online scheduling algorithm that computes an execution schedule $\mathcal{X}$ that achieves guaranteed space $S_P(\mathcal{X}) = O(S_1 P \lg P)$ and expected time $\mathrm{E}\left[T_P(\mathcal{X})\right] = O(T_1/P + T_\infty \lg P)$. Though this algorithm uses a $\lg P$ factor more space than the centralized algorithm, it can still achieve linear expected speedup — that is $\mathrm{E}\left[T_P(\mathcal{X})\right] = O(T_1/P)$ — provided the computation has sufficient average available parallelism — that is $T_1/T_\infty = \Omega(P \lg P)$. Furthermore, this algorithm is efficient in that on a PRAM or various low-latency, high-bandwidth fixed-connection networks, the overhead in computing the schedule is only a constant fraction of the execution time.

We also show that some nonstrictness can be allowed in an otherwise strict computation in a way that may improve performance, but does not adversely affect the time and space bounds.

Thesis Supervisor: Charles E. Leiserson

Title: Professor of Electrical Engineering and Computer Science

Contents

Acknowledgments

A few days ago, I was looking over my old research notes, and I was struck by just how much this research project has evolved. As I began this project, I was trying to understand how the space complexity of various parallel tree-search algorithms relates to the serial space used by a depth-first search and how probabilistic phenomenon affect the space usage of randomized, distributed tree-search algorithms. Over time, a model crystallized, questions formed, and eventually, answers gelled. When I described this evolution to my friend Arthur Lent, he said, "Now that's how research should be." Well, he wouldn't have said that if the project had not produced results, and the project would not have produced results had its evolution not been guided by someone who knew which directions to steer — someone who with each glimmer of understanding seemed to know where to look for the next insight that might turn the glimmer into vision. That someone is my advisor, Charles Leiserson.

As a grad student, I really only need one good project to keep me going, but a professor needs enough good ideas to keep a small flock of grad students busy with projects. Looking at the work of Charles' advisees past and present, it should come as no real surprise that I too have been the beneficiary of a great research project. I have truly enjoyed working on this project, and I look forward to continuing this line of research. Thank you, Charles.

Picking Charles as an advisor may be the smartest thing I've done since arriving in Boston, but the luckiest thing is surely my meeting the Bar-Yams: Miriam, Zvi, Yaneer, Aureet, and Sageet. The Bar-Yams are my landlords and my friends, but they have always treated me as family. They are among my favorite people on this planet, and its no wonder since the Bar-Yams are the nicest people on the planet — I've checked.

This research was supported in part by the Defense Advanced Research Projects Agency under Grant N00014-91-J-1698 and by a National Science Foundation Graduate Research Fellowship.

My sincere appreciation to:

- Tom Leighton: I learned most of the technique used in this research from courses taught by and papers written by Tom.
- Tom Cormen: If there is any quality in the figures and formatting in this thesis, I owe it to THC.
- William Ang, Be Hubbard, David Jones, Cheryl Patton, and Denise Sergent: Things work and social events happen because of their efforts.
- Bonnie Berger, Tom Cormen, Esther Jesurum, Michael Klugerman, Bradley Kuszmaul, Tom Leighton, Arthur Lent, and Greg Papadopoulos: for many insightful discussions.
- Bradley Kuszmaul, Atul Shrivastava, and Ethan Wolf: for careful proofreading.
- The faculty, staff, and students past and present of the Theory Group: for making my years here (so far) thoroughly enjoyable.

Finally, my thanks and appreciation goes to my family. Joanna, Brad, Benjamin, and Brandon — we may not be the closest family, but its nice to have some of the family here in Boston. Most of all, I thank my older brother, Michael. From my earliest recollections, Michael was my role model. I would not be who I am or where I am without Michael's enthusiasm for math and science and his patience in teaching me to program when I was still quite young.

Cambridge, Massachusetts
September 8, 1992

Robert D. Blumofe

Chapter 1

Introduction

In the course of investigating schemes for general purpose MIMD parallel computation, many diverse research groups have converged on multithreading as a dominant paradigm. As an example, modern dataflow systems [9, 11, 16, 24, 25, 26, 31, 32] partition the dataflow instructions into fixed groups called threads and arrange the instructions of each thread into a fixed sequential order at compile time. At run time, a scheduler employs dataflow concepts to dynamically order execution of the threads. Other systems have schedulers that dynamically order threads based on the availability of data in shared memory multiprocessors [1, 4, 13] or on the arrival of messages in message-passing multicomputers [2, 10, 20, 35].

Rapid execution of a multithreaded computation on a parallel computer requires exposing and exploiting parallelism in the computation by keeping enough threads concurrently active to keep the processors of the computer busy. If processors are busy most of the time, the execution schedule $\mathcal{X}$ of the computation exhibits linear speedup: the running time $T_P(\mathcal{X})$ with P processors is order P times faster than the optimal running time T_1 with 1 processor, that is, $T_P(\mathcal{X}) = O(T_1/P)$.

In attempting to expose parallelism, however, schedulers often end up exposing more parallelism than the computer can actually exploit, and since each active thread requires the use of a certain amount of memory, such schedulers can easily overrun the memory capacity of the machine [8, 12, 14, 30, 34]. To date, the space requirements of multithreaded computations have been managed with heuristics or not at all [7, 8, 12, 14, 17, 23, 30, 34]. In this thesis, we use algorithmic techniques to address the problem of managing storage for multithreaded

This thesis describes joint work with Charles E. Leiserson.

computations. Our goal is to develop scheduling algorithms that expose sufficient parallelism to obtain linear speedup, but without exposing so much parallelism that the space requirements become excessive.

We compare the amount of space $S_P(\mathcal{X})$ required by a P-processor execution schedule for a multithreaded computation with the space S_1 used by a space-optimal 1-processor execution. We wish to use as little space as possible, and we argue that a space-efficient execution schedule exhibits linear expansion of space, that is, $S_P(\mathcal{X}) = O(S_1 \cdot P)$.

Our first result shows that in general, it is not possible to achieve both linear speedup and linear expansion of space. We exhibit a multithreaded computation such that any execution schedule $\mathcal{X}$ that achieves P-processor execution time $T_P(\mathcal{X}) \leq T_1/\rho$ must use space at least $S_P(\mathcal{X}) \geq \frac{1}{4}(\rho - 1)\sqrt{T_1} + S_1$. For such a computation, even achieving a factor of 2 speedup ($\rho = 2$) requires space proportional to the square root of the serial execution time.

In order to cope with this negative result, we restrict our attention to the class of *strict* multithreaded computations. Intuitively, a strict computation is one in which no subroutine is called until all its parameters are available. We show that for any strict multithreaded computation and any number P of processors, there exists an execution schedule $\mathcal{X}$ that achieves time $T_P(\mathcal{X}) \leq T_1/P + T_\infty$, where T_∞ is a lower bound on execution time even for arbitrarily large numbers of processors, and space $S_P(\mathcal{X}) \leq S_1 P$. Such a schedule exhibits linear expansion of space and linear speedup, $T_P(\mathcal{X}) = O(T_1/P)$, provided the average available parallelism, which we define as T_1/T_∞, is at least proportional to P, that is, $T_1/T_\infty = \Omega(P)$. We demonstrate such schedules by exhibiting a simple centralized algorithm to compute them. We give a second, somewhat more efficient, algorithm that computes equally good execution schedules; this algorithm is online and should be practical for moderate numbers of processors, but its use of a centralized queue makes it inefficient for large numbers of processors.

To demonstrate an algorithm that is efficient even for large machines, we give a randomized, distributed, and online scheduling algorithm that achieves space expansion proportional to $P \lg P$ for any strict computation and linear expected speedup for any strict computation with average available parallelism at least proportional to $P \lg P$, that is, $T_1/T_\infty = \Omega(P \lg P)$.

This algorithm is efficient in that on a PRAM or various low-latency, high-bandwidth fixed-connection networks, the overhead in computing the schedule is only a constant fraction of the execution time.

We also show that some nonstrictness can be allowed in an otherwise strict computation in a way that may improve performance, but does not adversely affect the time and space bounds.

The remainder of this thesis is organized as follows. Chapter 2 develops a formal model of multithreaded computation and execution schedules. In Chapter 3 we characterize multithreaded computations with three parameters and prove some basic bounds relating these parameters to execution time and space. The lower bound for general multithreaded computations is presented in Chapter 4, and the upper bound for strict computations is presented in Chapter 5. Chapter 6 presents and analyzes a distributed scheduling algorithm for strict computations. In Chapter 7 we present a technique to allow nonstrictness without degrading the space and time bounds obtainable by a strict execution. Finally, in Chapter 8 we discuss some related work, and in Chapter 9 we conclude with some perspective on our results and some open problems.

Chapter 2

A model for multithreaded computation

This chapter defines the model of multithreaded computation that we use in this thesis. We also define what it means for a parallel computer to execute a multithreaded computation.

A multithreaded computation is composed of a set of threads, each of which is a sequential ordering of unit-size tasks. In Figure 2.1, for example, each shaded block is a thread with circles representing tasks and the horizontal edges, called *continue* edges, representing the sequential ordering. The tasks of a thread must execute in this sequential order from the first (leftmost) task to the last (rightmost) task. In order to execute a thread, we allocate for it a chunk of memory, called an *activation frame*, that the tasks 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 tasks at each step. An execution schedule depends on the particular multithreaded computation and the number of processors in the parallel computer. In any given step of an execution schedule, each processor either executes a single task or sits idle.

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 calling routine can operate concurrently with the called routine. We consider spawned threads to be children of the thread that did the spawning. In this way, threads are organized into a tree hierarchy as indicated in Figure 2.1 by the shaded edges, called *spawn* edges. Each spawn edge goes from a specific task, the task that actually does the spawn operation, in the parent thread to the first task of the child thread. When a thread executes its last task, it *terminates*.

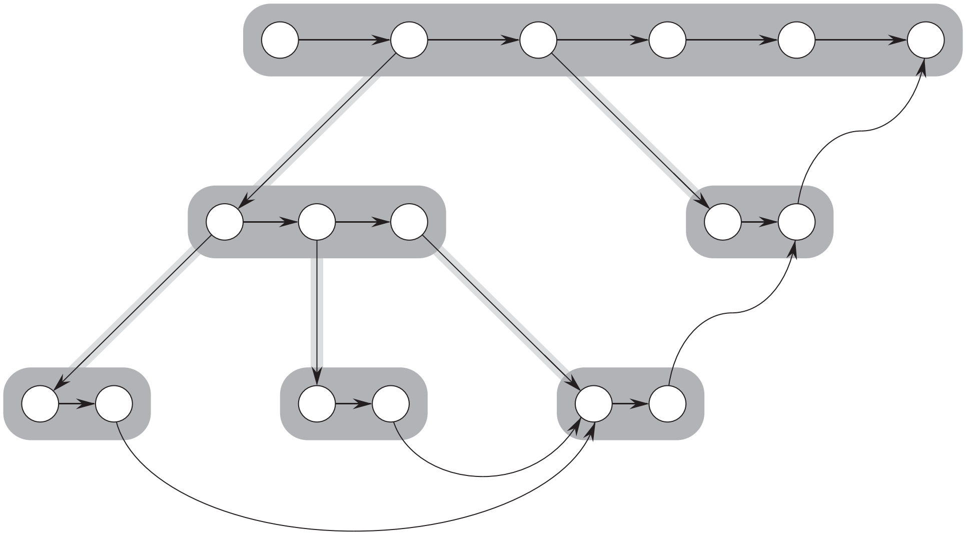

Figure 2.1: An example multithreaded computation. The tasks are partitioned into threads, represented by the shaded regions, and the tasks in each thread are compiled into a sequential order, represented by the continue edges shown horizontal in each thread. A task can spawn a thread, as shown by the shaded spawn edges, and this spawning organizes the threads into a tree hierarchy. The data dependency edges, shown by the curved edges, impose additional ordering constraints as required by producer/consumer relationships.

For an execution schedule to be valid, the task execution order must obey the constraints given by the edges of the computation. For example, before a task can execute, its predecessor — which connects to it via either a continue or spawn edge — must first execute.

There is one more kind of dependency that a valid execution schedule must respect. Consider a task that produces a data value that is consumed by another task. Such a producer/consumer relationship precludes the consuming task from executing until after the producing task. In order to enforce such orderings, we introduce *data dependency* edges as shown in Figure 2.1 by the curved edges. If the execution of a thread arrives at a consuming task before the producing task has executed, execution of the consuming thread cannot continue — the thread *stalls*. Once the producing task executes, the data dependency is *resolved*, and the consuming thread can proceed with its execution — the thread becomes *ready*.

We quantify the space used in executing a multithreaded computation in terms of activation frames. When a task spawns a thread, it allocates an activation frame for use by the newly spawned thread. Once a thread has been spawned and its frame has been allocated, we say the thread is *active.* Recall that at any time, an active thread can be either stalled or ready, but even if it stalls, its activation frame remains allocated. The thread remains active until it terminates; at that point its frame can be deallocated.

We make the simplifying assumption that a parent thread remains active until all its children terminate, and thus, a thread does not deallocate its activation frame until all its children's frames have been deallocated. Although this assumption is not strictly necessary, it gives the execution a natural structure, and it will simplify our analyses of space utilization. We also assume that the frames hold all the values used by the computation; there is no global storage available to the computation outside the frames. Therefore, the space used at a given time in executing a computation is the total size of all frames used by all active threads at that time, and the total space used in executing a computation is the maximum such value over the course of the execution.

It is important to note here the difference between what we are calling a multithreaded computation and a program. 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. Thus, what we are calling a thread actually represents a particular execution of a program thread. 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. We can think of a multithreaded computation as encapsulating both the program and the input data. The computation then reveals itself dynamically during execution.

An example

The multithreaded computation shown in Figure 2.2 contains 21 tasks, $v_1, v_2, \ldots, v_{21}$, and 5 threads, $\Gamma_1, \Gamma_2, \ldots, \Gamma_5$. Execution begins with the root thread Γ_1 active and ready. Thread Γ_1 has activation frame size $\mathcal{F}(\Gamma_1) = 3$, so the execution begins with 3 units of space in use. At the first step of the execution, a processor executes task v_1. At the end of the first step, Γ_1 is

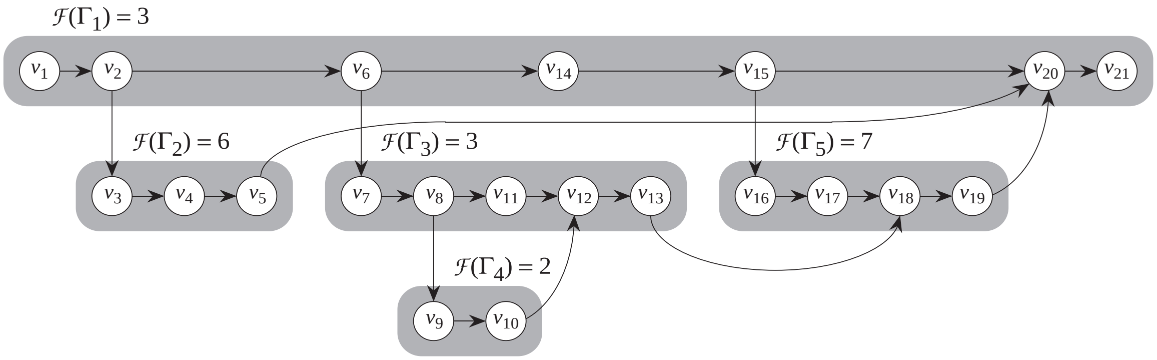

Figure 2.2: A multithreaded computation. This computation has 21 tasks, $v_1, v_2, \ldots, v_{21}$, and 5 threads, $\Gamma_1, \Gamma_2, \ldots, \Gamma_5$, with activation frame sizes, $\mathcal{F}(\Gamma_1) = 3$, $\mathcal{F}(\Gamma_2) = 6$, $\mathcal{F}(\Gamma_3) = 3$, $\mathcal{F}(\Gamma_4) = 2$, and $\mathcal{F}(\Gamma_5) = 7$.

still the only active (and ready) thread, and therefore, at the second step, a processor executes task v_2. Task v_2 spawns a child thread Γ_2 with activation frame size $\mathcal{F}(\Gamma_2) = 6$. Consequently, the second step ends with $3 + 6 = 9$ units of space in use and both Γ_1 and Γ_2 active and ready. Then if the parallel machine executing this computation has at least two processors, task v_6 from Γ_1 and task v_3 from Γ_2 can execute concurrently during the third step. Executing task v_6 spawns another thread which further increases the amount of space in use. Eventually, when task v_5 executes, thread Γ_2 terminates and decreases the amount of space in use. Furthermore, executing v_5 resolves the data dependency (v_5, v_{20}). When the execution of thread Γ_1 reaches v_{20}, the thread stalls until both data dependencies (v_5, v_{20}) and (v_{19}, v_{20}) resolve.

Figures 2.3 and 2.4 show two different 2-processor execution schedules for the computation of Figure 2.2. The schedule of Figure 2.3 takes 14 time steps and 13 units of space. The schedule of Figure 2.4 takes 15 time steps and 21 units of space; for a period of time during the execution of this schedule, every thread in the computation is active.

Time	Tasks executed	Active threads	Space in use
0		$\enclose{circle}{\Gamma_1}$	3
1	v_1	$\enclose{circle}{\Gamma_1}$	3
2	v_2	$\enclose{circle}{\Gamma_1}$ $\enclose{circle}{\Gamma_2}$	9
3	v_3 v_6	$\enclose{circle}{\Gamma_1}$ $\enclose{circle}{\Gamma_2}$ $\enclose{circle}{\Gamma_3}$	12
4	v_4 v_7	$\enclose{circle}{\Gamma_1}$ $\enclose{circle}{\Gamma_2}$ $\enclose{circle}{\Gamma_3}$	12
5	v_5 v_8	$\enclose{circle}{\Gamma_1}$ $\enclose{circle}{\Gamma_3}$ $\enclose{circle}{\Gamma_4}$	8
6	v_9 v_{11}	$\enclose{circle}{\Gamma_1}$ Γ_3 $\enclose{circle}{\Gamma_4}$	8
7	v_{10} v_{14}	$\enclose{circle}{\Gamma_1}$ $\enclose{circle}{\Gamma_3}$	6
8	v_{12} v_{15}	Γ_1 $\enclose{circle}{\Gamma_3}$ $\enclose{circle}{\Gamma_5}$	13
9	v_{13} v_{16}	Γ_1 $\enclose{circle}{\Gamma_5}$	10
10	v_{17}	Γ_1 $\enclose{circle}{\Gamma_5}$	10
11	v_{18}	Γ_1 $\enclose{circle}{\Gamma_5}$	10
12	v_{19}	$\enclose{circle}{\Gamma_1}$	3
13	v_{20}	$\enclose{circle}{\Gamma_1}$	3
14	v_{21}		

Figure 2.3: An execution schedule for the computation illustrated in Figure 2.2 with two processors. Each row represents one time step of the computation as indicated in the first column. The second column lists the tasks that execute at the associated time step. The third column lists the threads that are active at the end of the associated time step; threads that are also ready are shown circled. The last column shows how much space is in use at the end of the associated time step. This execution takes 14 time steps and 13 units of space.

Time	Tasks executed		Active threads					Space in use
0			$\enclose{circle}{\Gamma_1}$					3
1	v_1		$\enclose{circle}{\Gamma_1}$					3
2	v_2		$\enclose{circle}{\Gamma_1}$	$\enclose{circle}{\Gamma_2}$				9
3	v_3	v_6	$\enclose{circle}{\Gamma_1}$	$\enclose{circle}{\Gamma_2}$	$\enclose{circle}{\Gamma_3}$			12
4	v_4	v_{14}	$\enclose{circle}{\Gamma_1}$	$\enclose{circle}{\Gamma_2}$	$\enclose{circle}{\Gamma_3}$			12
5	v_7	v_{15}	Γ_1	$\enclose{circle}{\Gamma_2}$	$\enclose{circle}{\Gamma_3}$	$\enclose{circle}{\Gamma_5}$		19
6	v_8	v_{16}	Γ_1	$\enclose{circle}{\Gamma_2}$	$\enclose{circle}{\Gamma_3}$	$\enclose{circle}{\Gamma_4}$	$\enclose{circle}{\Gamma_5}$	21
7	v_{11}	v_{17}	Γ_1	$\enclose{circle}{\Gamma_2}$	Γ_3	$\enclose{circle}{\Gamma_4}$	Γ_5	21
8	v_5	v_9	Γ_1	Γ_3	$\enclose{circle}{\Gamma_4}$	Γ_5		15
9	v_{10}		Γ_1	$\enclose{circle}{\Gamma_3}$	Γ_5			13
10	v_{12}		Γ_1	$\enclose{circle}{\Gamma_3}$	Γ_5			13
11	v_{13}		Γ_1	$\enclose{circle}{\Gamma_5}$				10
12	v_{18}		Γ_1	$\enclose{circle}{\Gamma_5}$				10
13	v_{19}		$\enclose{circle}{\Gamma_1}$					3
14	v_{20}		$\enclose{circle}{\Gamma_1}$					3
15	v_{21}							

Figure 2.4: Another execution schedule for the computation illustrated in Figure 2.2 with two processors. This execution takes 15 time steps and 21 units of space.

Chapter 3

Time and space

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

The two time parameters are based on the underlying graph structure of the multithreaded computation. If we ignore the shading in Figure 2.1 that organizes tasks into threads, our multithreaded computation is just a directed, acyclic graph, or *dag*. We define the *work* of the computation to be the total number of tasks and the *computation depth* to be the length of a longest directed path in the dag. For example, the computation of Figure 2.1 has work 17 and computation depth 10, and the computation of Figure 2.2 has work 21 and computation depth 13.

We quantify and bound the execution time of a computation on a P-processor parallel computer in terms of the computation's work and depth. For a given computation, let $T_P(\mathcal{X})$ denote the time to execute the computation with P processors using execution schedule $\mathcal{X}$, and let

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

denote minimum time execution with P processors — the minimum being taken over all valid execution schedules for the computation. Then T_1 is the work of the computation, since a 1-processor computer can only execute one task at each step, and T_∞ is the computation depth, since even with arbitrarily many processors, each task on a path must execute serially.

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 tasks per step, and since the computation has T_1 tasks, $T_P \geq T_1/P$. And, of course, we also have $T_P \geq T_\infty$. Brent's Theorem [5, Lemma 2] yields the bound $T_P \leq T_1/P + T_\infty$. The following theorem extends Brent's Theorem minimally to show that this upper bound on T_P can be obtained by *greedy schedules*: those in which at each step of the execution, if at least P tasks are ready, then P tasks execute, and if fewer than P tasks are ready, then all execute; both of the schedules shown in Figures 2.3 and 2.4 are greedy.

Theorem 1 *For any multithreaded computation with work T_1 and computation depth T_∞, for any number P of processors, any greedy execution schedule $\mathcal{X}$ achieves $T_P(\mathcal{X}) \leq T_1/P + T_\infty$.*

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_∞. Consider a greedy execution schedule $\mathcal{X}$ where the set of tasks executed at time i, for $i = 1, 2, \ldots, k$, is denoted $\mathcal{E}_i$, with $k = T_P(\mathcal{X})$. The $\mathcal{E}_i$ 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 $i = 1, 2, \ldots, k$, we have $V_i = V_{i-1} - \mathcal{E}_i$ and G_i is the subgraph of G_{i-1} induced by V_i. In other words, G_i is obtained from G_{i-1} by removing from G_{i-1} all the tasks that are executed by $\mathcal{X}$ at step i and all edges incident on these tasks. 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 i according to $|\mathcal{E}_i|$. Consider a step i with $|\mathcal{E}_i| = P$. In this case, $|V_i| = |V_{i-1}| - P$, so since $|V| = T_1$, there can be at most $\lfloor T_1/P \rfloor$ such steps. Now consider a step i with $|\mathcal{E}_i| < P$. In this case, since $\mathcal{X}$ is greedy, $\mathcal{E}_i$ must contain every vertex of G_{i-1} with in-degree 0. Therefore, the length of a longest path in G_i is one less than the length of a longest path in G_{i-1}. Since the length of a longest path in G is T_∞, there can be no more than T_∞ steps i with $|\mathcal{E}_i| < P$.

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

Theorem 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 $T_1/P + T_\infty \leq 2\max\{T_1/P, T_\infty\}$ and $T_P \geq \max\{T_1/P, T_\infty\}$. Second, Theorem 1 tells us when we can obtain *linear parallel speedup*, that is, when we can find an execution schedule $\mathcal{X}$ such that $T_P(\mathcal{X}) = \Theta(T_1/P)$. Specifically, when the number P of processors is no more than the *average available parallelism* T_1/T_∞, then $T_1/P \geq T_\infty$, which implies that for a greedy schedule $\mathcal{X}$, we have $T_P(\mathcal{X}) \leq 2T_1/P$. 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 $T_P \geq T_\infty$.

These results on dag scheduling have been known for many years. A multithreaded computation, however, adds further structure to the dag: the partitioning of tasks into threads. This additional structure allows us to quantify the space used in executing a multithreaded computation. Once we have quantified space usage, we will look back at Theorem 1 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*.

We shall focus on a space parameter for a multithreaded computation which is based on the tree structure of threads. If we collapse each thread into a single node and consider just the spawn edges, the multithreaded computation becomes a rooted tree with the spawn edges as child pointers. We call this tree the *activation tree*. We define the *activation depth* of a thread to be the sum of the sizes of the activation frames of all its ancestors, including itself. The *activation depth* of a multithreaded computation is the maximum activation depth of any thread. For example, in the computation of Figure 2.2, thread Γ_4 has activation depth 8, and the computation has activation depth 10, since the deepest thread Γ_5 has activation depth 10.

We shall have occasion to consider subcomputations and subcomputation activation depth. A subcomputation is the portion of a computation rooted at a given thread in the activation tree, and the activation depth of a subcomputation is the activation depth of the subcomputation when considered in isolation as a multithreaded computation. For example, in the computation of Figure 2.2, the subcomputation rooted at thread Γ_3 consists of 7 tasks, $v_7, v_8, \ldots, v_{13}$, and 2 threads, Γ_3 and Γ_4, and has activation depth $3 + 2 = 5$.

We shall denote the space required by a P-processor execution schedule $\mathcal{X}$ of a multithreaded computation by $S_P(\mathcal{X})$. Recall that $S_P(\mathcal{X})$ is just the maximum, over all steps in $\mathcal{X}$, of the sum of the sizes of the activation frames of the active threads at that step. Since we can always simulate a P-processor execution with a 1-processor execution that uses no more space, we have $S_1(\mathcal{X}) \leq S_P(\mathcal{X})$. The minimum space used by any execution with any number of processors is therefore $S_1 = \min_{\mathcal{X}} S_1(\mathcal{X})$.

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

Theorem 2 *Let $\mathcal{A}$ be the activation depth of a multithreaded computation, and let $\mathcal{X}$ be a P-processor execution schedule of the computation. Then $S_P(\mathcal{X}) \geq \mathcal{A}$, and hence, $S_1 \geq \mathcal{A}$.*

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

Given the lower bound of activation depth on the space used by a P-processor schedule, it is natural to ask whether the activation 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 data dependencies that force all of them to be simultaneously active in any execution schedule. For the class of *depth-first* computations, however, space equal to the activation depth can be achieved by a 1-processor schedule.

A *depth-first* computation is a multithreaded computation in which a left-to-right depth-first search of tasks in the activation tree always visits all the tasks on which a given task depends before it visits the given task. In fact, this depth-first search produces a 1-processor execution schedule which is just the familiar stack-based execution: The serial depth-first execution begins with the root thread and executes its tasks until it either spawns a child thread or terminates. If the thread spawns a child, the parent thread is put aside to be resumed only after the child thread terminates; the scheduler then begins work on the child, executing the child until it either spawns or terminates. For the computation of Figure 2.2, the 1-processor execution schedule that executes tasks in the order $v_1, v_2, v_3, \ldots, v_{20}, v_{21}$ is the serial depth-first schedule.

Theorem 3 *For any depth-first computation, $S_1 = \mathcal{A}$.*

Proof: At any time in a serial depth-first execution of the computation, the set of active threads always forms a path from the root. Therefore, the space required is just the activation depth of the computation. By Theorem 2, $S_1 \geq \mathcal{A}$, and thus the the space used is the minimum possible. ■

We now turn our attention to determining how much space $S_P(\mathcal{X})$ 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 required by an optimal 1-processor schedule. Of course, we can always ignore $P - 1$ of the processors and obtain the same space bounds, and therefore, our goal is to use small space while obtaining linear speedup.

Even for depth-first computations, a P-processor schedule may use nearly P times the space of a 1-processor schedule. We exhibit a depth-first computation with activation depth $\mathcal{A} = S_1$ that for any number $P \leq T_1/T_\infty$ of processors, requires space nearly $S_1 P$ in order to achieve linear parallel speedup. In the computation, the root thread, which we refer to as the *loop*, spawns many children, and each child thread is the root of a large subcomputation, which we refer to as an *iteration*. The root thread has an activation frame of size 1, and each iteration has activation depth $S_1 - 1$. See Figure 3.1. In addition, data dependencies force a serial ordering on the tasks within each iteration, but there are no data dependencies between tasks in different iterations. In other words, the entire computation has no available parallelism within an iteration; parallelism can only be realized by the concurrent execution of multiple iterations. Executing P iterations concurrently, uses space $P(S_1 - 1) + 1$ which is nearly $S_1 P$.

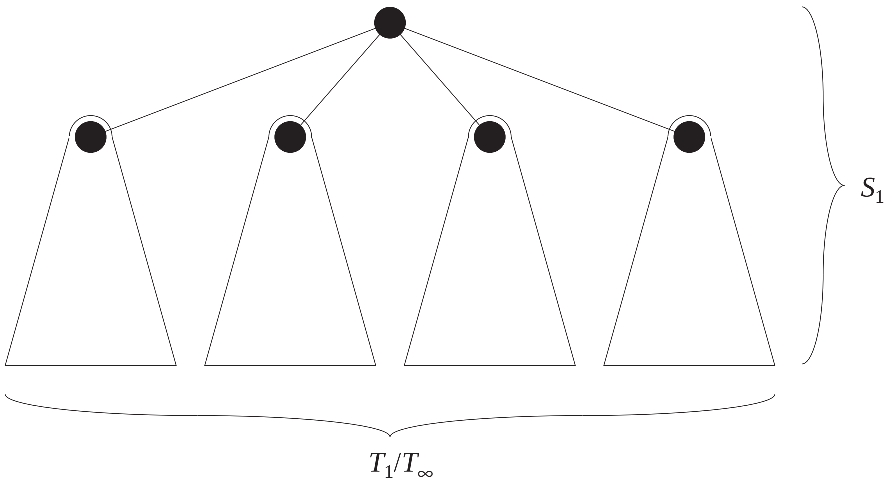

Figure 3.1: The activation tree of a multithreaded computation for which any execution schedule $\mathcal{X}$ requires space $S_P(\mathcal{X}) = \Omega(S_1 P)$ in order to achieve linear speedup. The root thread is a loop and each child thread is the root of a subcomputation that forms an iteration. The data dependencies in each iteration (not shown) link the tasks of the iteration into a sequential order, so there is no parallelism within the iteration. Between iterations, however, there are no data dependencies, so multiple iterations can be executed concurrently. The average available parallelism T_1/T_∞ equals the number of iterations. Therefore, for any number $P \leq T_1/T_\infty$ of processors, there is an execution schedule $\mathcal{X}$ (any greedy schedule for example) that achieves $T_P(\mathcal{X}) = \Theta(T_1/P)$ and space $S_P(\mathcal{X}) = P(S_1 - 1) + 1 = \Theta(S_1 P)$.

Thus, for any number $P \leq T_1/T_\infty$ of processors, this computation has an execution schedule $\mathcal{X}$ (any greedy schedule, for example) that achieves linear speedup, $T_P(\mathcal{X}) = \Theta(T_1/P)$, at the cost of space $S_P(\mathcal{X}) = \Theta(S_1 P)$.

In fact, a P-processor schedule that uses only P times the space of a single processor is arguably efficient, since on average, each of the P processors only needs as much memory as is used by the 1 processor. We would, of course, like to do better, but an expansion in space that is linear in the number of processors, while achieving linear speedup, is quite good, since the time-space product is bounded by a constant:

$$T_P(\mathcal{X}) S_P(\mathcal{X}) = O(T_1 S_1) \ .$$

We shall show in Chapter 4 that achieving linear speedup and linear expansion of space simultaneously is impossible in general, even for depth-first computations. For a restricted class of computations that we call *strict*, however, Chapter 5 shows that one can achieve both.

To summarize, we can parameterize a multithreaded computation with three measures:

- T_1 denotes the work of the computation,
- T_∞ denotes its computation depth,
- $\mathcal{A}$ denotes its activation depth.

For depth-first computations, $S_1 = \mathcal{A}$. For any number $P = O(T_1/T_\infty)$ of processors, we would like to find an execution schedule $\mathcal{X}$ with the following time and space bounds:

- $T_P(\mathcal{X}) = O(T_1/P)$,
- $S_P(\mathcal{X}) = O(S_1 P)$.

Chapter 4

Lower bound

In this chapter we show that there exist multithreaded computations for which no execution schedule can achieve both linear speedup and linear expansion of space. In particular, for any amount of serial space $\mathcal{S}$ and any (reasonably large) serial execution time $\mathcal{T}$, we can exhibit a depth-first multithreaded computation with work $T_1 = \mathcal{T}$ and activation depth $\mathcal{A} = \mathcal{S}$ but with provably bad time/space tradeoff characteristics. Being depth-first, we know from Theorem 3 that our computation can be executed using serial space $S_1 = \mathcal{A}$. Furthermore, we know from Theorem 1 that for any number P of processors, any greedy P-processor execution schedule $\mathcal{X}$ achieves $T_P(\mathcal{X}) \leq T_1/P + T_\infty$. Our computation has computation depth T_∞ approximately $\sqrt{T_1}$, and consequently, for $P = O(\sqrt{T_1})$, a greedy schedule $\mathcal{X}$ yields $T_P(\mathcal{X}) = O(T_1/P)$ — linear speedup. We show, however, that for this computation, any schedule achieving $T_P(\mathcal{X}) = O(T_1/P)$ must use space $S_P(\mathcal{X}) = \Omega(\sqrt{T_1}(P-1))$. Of course, $\sqrt{T_1}$ may be much larger than S_1, hence, this space bound is nowhere near linear in its space expansion.

We construct a multithreaded computation having this poor time/space performance by placing tasks that are computationally deep into the same portion of the computation as tasks that are computationally shallow. If we look at just the dag structure of the computation, it appears, from a distance, as shown in Figure 4.1 — the dag in Figure 4.1 is just missing a few of the tasks and edges that organize the computation into a tree hierarchy. The dag consists of m (a value we will specify later) components $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{m-1}$ that we call *jobs*. From this dag, we see that with any number $P \leq m$ of processors, we can obtain linear speedup by simultaneously executing P jobs. Doing so, however, uses up lots of memory. To execute a job $\mathcal{C}_i$, we begin with a group of computationally shallow tasks called *headers* (see Figure 4.1). Each header is

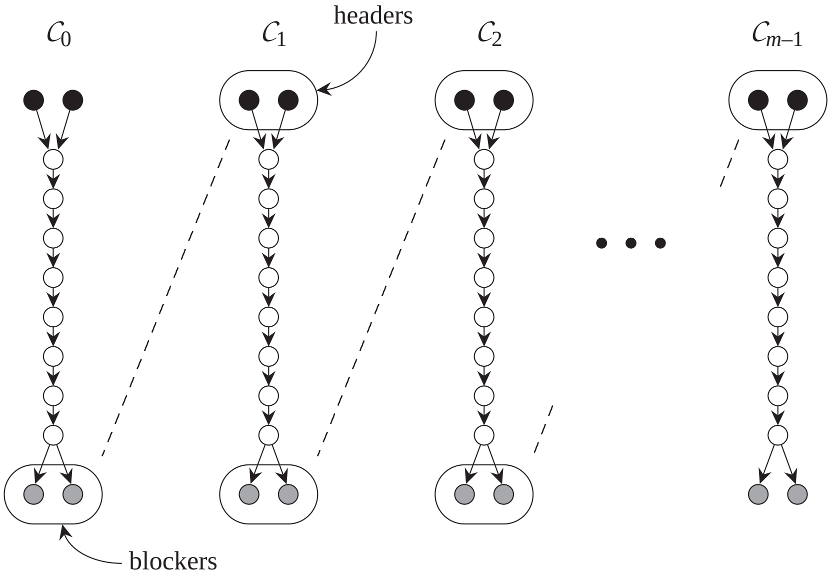

Figure 4.1: The tasks in the leaf threads are organized into m jobs, $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{m-1}$. The black header tasks have shallow computation depth. The white tasks form the trunk of the job. The grey blocker tasks have deep computation depth.

part of a separate subcomputation with fairly large activation depth, so to execute a header task we must begin execution of its associated subcomputation by allocating the necessary activation frames. Each of these subcomputations also contains a computationally deep task, called a *blocker* (see Figure 4.1), from the previous job $\mathcal{C}_{i-1}$. Therefore, these subcomputations cannot complete, and the associated memory cannot be deallocated until the blockers from the previous job execute. But in order to achieve speedup, jobs must execute concurrently, and consequently, the headers must execute early and the blockers must execute late. Therefore, in this scenario, many subcomputations begin early, but cannot finish until late, hence the heavy demands on storage.

Theorem 4 *For any amount of serial space $\mathcal{S} \geq 4$ and serial time $\mathcal{T} \geq 16\mathcal{S}^2$, there exists a depth-first multithreaded computation with work $T_1 = \mathcal{T}$, computation depth $T_\infty \leq 8\sqrt{T_1}$, and activation depth $\mathcal{A} = \mathcal{S}$, such that for any number P of processors and any value ρ in the range $1 \leq \rho \leq \frac{1}{8}T_1/T_\infty$, if $\mathcal{X}$ is a valid P-processor execution schedule that achieves $T_P(\mathcal{X}) \leq T_1/\rho$, then $S_P(\mathcal{X}) \geq \frac{1}{4}(\rho - 1)\sqrt{T_1} + S_1$.*

Proof: To exhibit a depth-first multithreaded computation with work T_1, computation depth T_∞, and activation depth $\mathcal{A} = S_1$, we first consider the dag structure of the computation. If we look at just the tasks in the leaf threads and ignore a few of the edges, the dag appears as in Figure 4.1. The tasks are organized into

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

(nearly) separate components $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{m-1}$ that we call *jobs.*[1] Each job begins with

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

tasks that we call *headers.* After the headers, each job contains

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

tasks organized into a chain that we call the *trunk.* There are no dependencies between the headers, but the first task of the trunk cannot execute until after all the headers. At the end of each job, there are λ *blockers.* Each job, therefore, consists of $2\lambda + \nu = 2(\sqrt{T_1}/S_1) + 6\sqrt{T_1}$ tasks. Since there are $m = \sqrt{T_1}/8$ jobs, the total number of tasks accounted for by the m jobs 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) $\frac{3}{16}T_1$ tasks form the parts of the computation not shown in Figure 4.1.

When we consider how the tasks of each job are organized into the threads of the computation, we will exhibit an organization such that each header task is part of a separate subcomputation with activation depth at least $\frac{1}{2}S_1$. This organization will also be such that each of these subcomputations contains a blocker task from a different job. In particular, each job $\mathcal{C}_i$, for $i = 1, \ldots, m-1$, has each of its header tasks in a subcomputation that also contains a blocker task of the previous job $\mathcal{C}_{i-1}$. For each such subcomputation, the blocker task is placed

[1] In what follows, we refer to a number x of objects (such as tasks) 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.

to ensure that the subcomputation cannot complete until the blocker task executes. Therefore, from the time the header task of job $\mathcal{C}_i$ executes until the time the blocker task of job $\mathcal{C}_{i-1}$ executes, all of the (at least) $\frac{1}{2}S_1$ space used by the subcomputation remains active. Furthermore, if all of the headers of $\mathcal{C}_i$ execute before any of the blockers of $\mathcal{C}_{i-1}$, then during the intervening time period, λ of these subcomputations are active, and these active subcomputations take up at least $\frac{1}{2}S_1\lambda = \frac{1}{2}\sqrt{T_1}$ space. We will show that in fact, this space consuming scenario must occur in any execution schedule that achieves any amount of parallel speedup.

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

Now consider two jobs, $\mathcal{C}_i$ and $\mathcal{C}_{i-1}$, and suppose $t_i^{(\mathrm{s})} < t_{i-1}^{(\mathrm{f})}$; this is the scenario we described as using at least $\frac{1}{2}\sqrt{T_1}$ 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 job $\mathcal{C}_i$ is *exposed*, and we let $\tau_i = t_{i-1}^{(\mathrm{f})} - t_i^{(\mathrm{s})}$ denote the amount of time job $\mathcal{C}_i$ is exposed. See Figure 4.2. If $t_i^{(\mathrm{s})} \geq t_{i-1}^{(\mathrm{f})}$ then job $\mathcal{C}_i$ is never exposed and we let $\tau_i = 0$. As we have seen, over the time interval during which a job is exposed, it uses at least $\frac{1}{2}\sqrt{T_1}$ space. We will show that in order to achieve speedup ρ — that is $T_P(\mathcal{X}) \leq T_1/\rho$ — there must be some time step during the execution at which at least $\lceil \frac{3}{4}\rho \rceil - 1$ jobs are exposed.

If schedule $\mathcal{X}$ is such that $T_P(\mathcal{X}) \leq T_1/\rho$, then we must have $t_{m-1}^{(\mathrm{f})} - t_0^{(\mathrm{s})} \leq T_1/\rho$, and we can expand this inequality out as

$$\begin{aligned}
T_1/\rho &\geq t_{m-1}^{(\mathrm{f})} - t_0^{(\mathrm{s})} \\
&= t_0^{(\mathrm{f})} + \sum_{i=1}^{m-1} (t_i^{(\mathrm{f})} - t_{i-1}^{(\mathrm{f})}) - t_0^{(\mathrm{s})} \\
&= t_0^{(\mathrm{f})} - t_0^{(\mathrm{s})} + \sum_{i=1}^{m-1} \left((t_i^{(\mathrm{f})} - t_i^{(\mathrm{s})}) + (t_i^{(\mathrm{s})} - t_{i-1}^{(\mathrm{f})}) \right) \\
&= \sum_{i=0}^{m-1} (t_i^{(\mathrm{f})} - t_i^{(\mathrm{s})}) - \sum_{i=1}^{m-1} (t_{i-1}^{(\mathrm{f})} - t_i^{(\mathrm{s})}). \qquad (4.1)
\end{aligned}$$

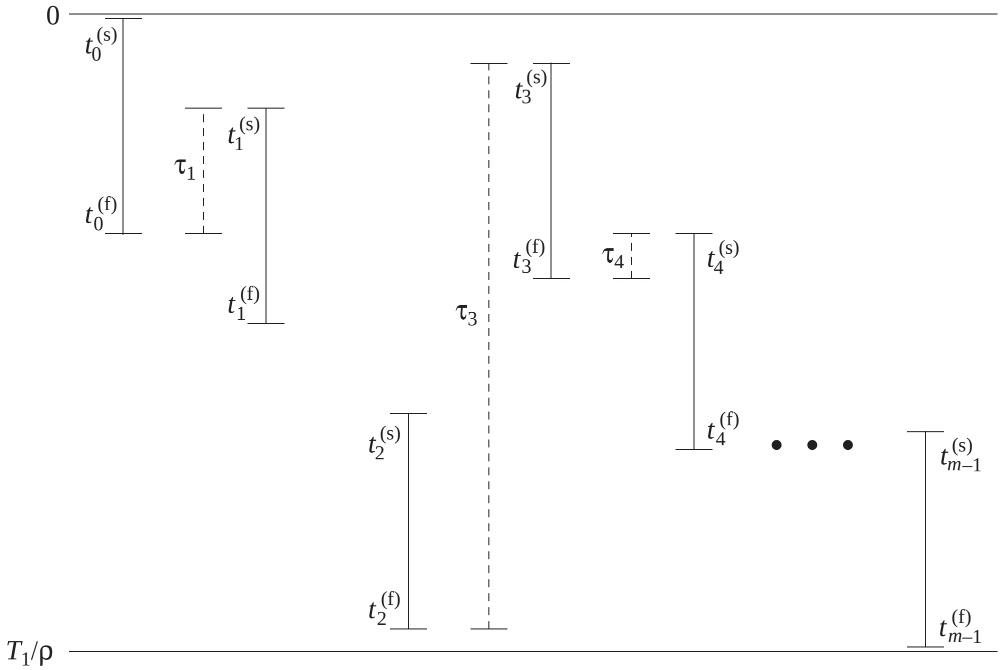

Figure 4.2: Scheduling the execution of the jobs. A solid vertical interval from $t_i^{(s)}$ to $t_i^{(f)}$ indicates the time during which the trunk of job $\mathcal{C}_i$ is being executed. When $t_i^{(s)} < t_{i-1}^{(f)}$, we can define an interval, shown dashed, of length $\tau_i = t_{i-1}^{(f)} - t_i^{(s)}$, during which job $\mathcal{C}_i$ is exposed.

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

$$\sum_{i=0}^{m-1} (t_i^{(f)} - t_i^{(s)}) \geq m\nu. \tag{4.2}$$

Considering the second sum of Inequality (4.1), when $t_{i-1}^{(f)} > t_i^{(s)}$ (so $\mathcal{C}_i$ 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,

$$\sum_{i=1}^{m-1} (t_{i-1}^{(f)} - t_i^{(s)}) \leq \sum_{i=1}^{m-1} \tau_i. \tag{4.3}$$

Substituting Inequality (4.2) and Inequality (4.3) back into Inequality (4.1), we obtain

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

from which

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

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

$$\sum_{t=0}^{T_1/\rho} \mathit{exposed}(t) = \sum_{i=i}^{m-1} \tau_i.$$

Then the average number of exposed jobs per time step is

$$\begin{aligned}
\frac{1}{T_1/\rho} \sum_{t=0}^{T_1/\rho} \mathit{exposed}(t) &= \frac{1}{T_1/\rho} \sum_{i=1}^{m-1} \tau_i \\
&\geq \frac{1}{T_1/\rho} (m\nu - T_1/\rho) \\
&= \frac{m\nu}{T_1}\rho - 1 \\
&= \frac{3}{4}\rho - 1
\end{aligned}$$

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,

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

Now recalling that each exposed job uses space $\frac{1}{2}\sqrt{T_1}$, we have

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

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 tasks of each job into a depth-first multithreaded computation with work T_1, computation depth $T_\infty \leq 8\sqrt{T_1}$, and activation depth $\mathcal{A} = S_1$ in such a way that for each job, each header task is placed in a subcomputation with a

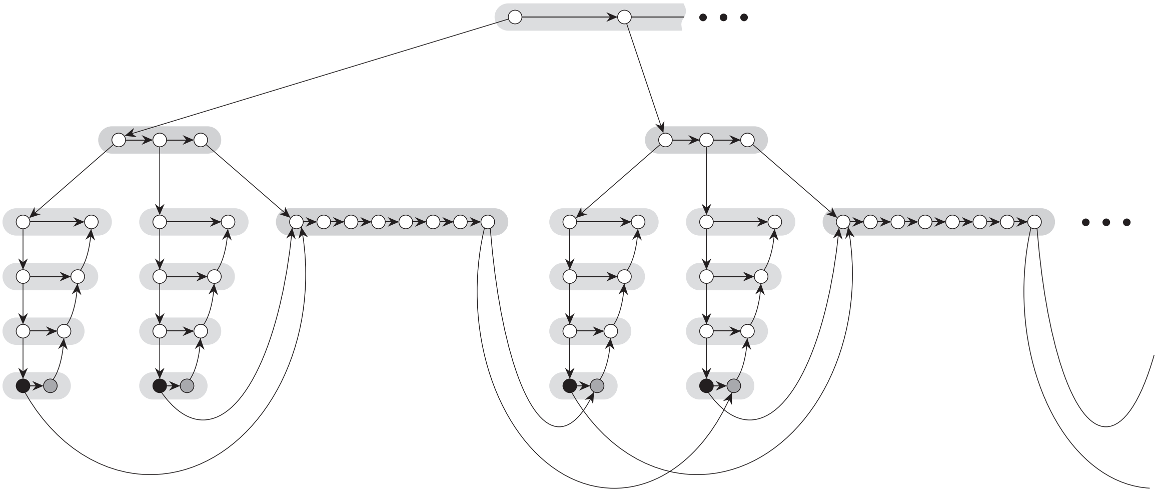

Figure 4.3: Laying out the jobs into the threads of a multithreaded computation. In this example, each activation frame has unit size so $\mathcal{A} = 6$. Also, in this example $\lambda = 2$, $\nu = 8$, and only the first 2 out of the m tasks in the root thread are shown. Each task of the root thread spawns a child, and each child thread contains $\lambda + 1 = 3$ tasks; the first λ of these spawn a child thread which is the root of a subcomputation with activation depth $\mathcal{A} - 2 = 4$, and the last one spawns a leaf thread with the $\nu = 8$ trunk tasks of a single job.

blocker task from the previous job and that each such subcomputation has activation depth at least $S_1/2$. 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 4.3.

For the multithreaded computation of Figure 4.3, the root thread contains m tasks, each of which spawns a child thread. Each child thread contains $\lambda + 1$ tasks; the first λ of these spawn a child thread which is the root of a subcomputation with activation depth $S_1 - 2 \geq S_1/2$ (since $S_1 \geq 4$), and the last one spawns a leaf thread with the ν trunk tasks of a single job. Each of these subcomputations contains a single header from one job and a single blocker from the previous job (except in the case of the first group of λ) as shown in Figure 4.3. The header and blocker in a subcomputation are organized such that in order to execute the header, all $S_1 - 2$ of the threads in the subcomputation must be made active, and none of them can terminate until the blocker executes. We can verify from Figure 4.3 and from the given values of m, λ, and ν 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 tasks evenly among the threads that contain the trunk of each job (thereby increasing ν by a bit). Also, we can verify that $T_\infty \leq 8\sqrt{T_1}$. Finally, looking at Figure 4.3 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 restriction to the model while still obtaining the same result. For example, some real multithreaded systems require limits on the number of tasks in a thread, data dependencies that only go to the first task of a thread, limited fan-in for data 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 4 holds even for multithreaded computations with any or all of these restrictions.

Theorem 4 tells us that for any amount of serial space $\mathcal{S}$ and any (reasonably) large serial execution time $\mathcal{T}$, there exists a multithreaded computation that can be executed serially in the given amount of time and space, has sufficient average available parallelism to achieve linear speedup over a wide range of numbers of processors, but in order to achieve any speedup at all, requires (potentially) extreme amounts of space. For example, in order to achieve linear speedup when the number of processors is close to the average available parallelism, such a computation requires space proportional to T_1 — the serial execution time. Even to achieve speedup of 2 ($\rho = 2$), such a computation requires space proportional to $\sqrt{T_1}$ — not quite T_1, but still potentially huge compared to S_1.

There are actually many ways of stating a lower bound as in Theorem 4, but they all come down to the same thing: There exist multithreaded computations with arbitrary serial execution time and space and with arbitrarily large amounts of average available parallelism, such that achieving any amount of speedup ranging from 1 (no speedup) up to the average available parallelism requires space that ranges from the serial space up to nearly the serial time correspondingly.

Chapter 5

Scheduling algorithms for strict multithreaded computations

Given a multithreaded computation, a scheduling algorithm for a P-processor parallel computer must compute a valid P-processor execution schedule. In computing such a schedule, the algorithm does not know the entire computation; the computation actually unfolds dynamically during the course of execution, and consequently, the scheduling algorithm must be online. At any given time during the execution, the scheduler has a set of active threads some of which are ready and some of which are stalled. There might be some extra information attached to each thread that the scheduling algorithm can use in deciding which ready threads get executed by which processors, but the scheduler cannot know about the structure of the portion of the computation not yet executed.

Besides being able to compute an efficient execution schedule, we would like the scheduling algorithm itself to be efficient. In computing the execution schedule, the algorithm incurs costs that we can broadly classify into three categories: queueing costs, synchronization costs, and communication costs. The scheduling algorithm maintains active threads in one or more queues. By enqueuing and dequeuing threads over the course of execution, the scheduler incurs queueing costs. If the scheduling algorithm requires the use of any shared data or global values, it incurs synchronization costs. Suppose that at some time during the computation, the scheduling algorithm decides that a processor p should execute a task from thread Γ, and then at some later time, the scheduler decides that a different processor $p' \neq p$ should execute a task from the same thread Γ. In this case, some information about Γ, possibly the entire activation frame, must be moved from processor p to processor p'. In doing so, the scheduling algorithm incurs

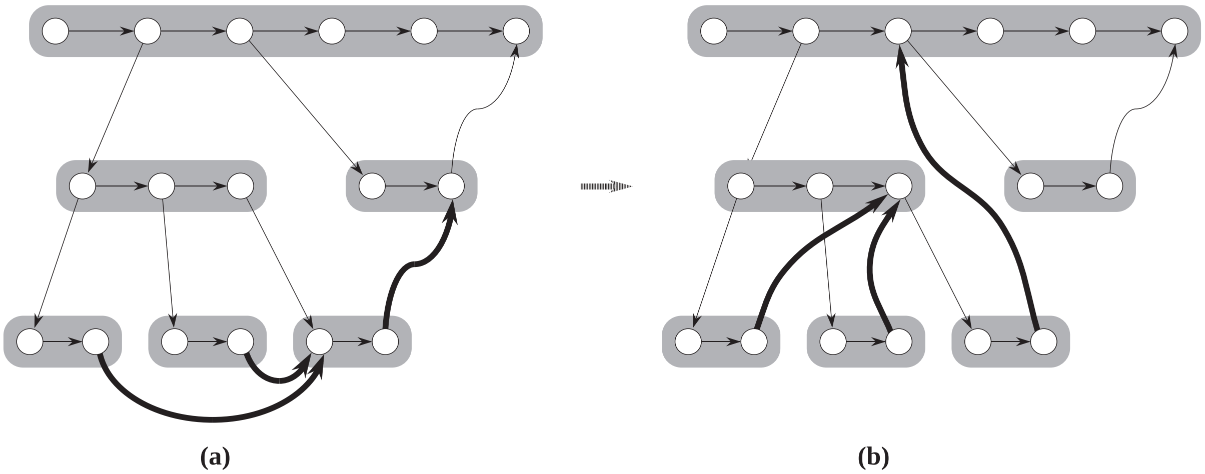

Figure 5.1: **(a)** This multithreaded computation is nonstrict since it has data dependencies, shown bold, that go to non-ancestor threads. **(b)** If we replace the offending data dependencies with new ones, shown bold, we obtain a strict computation since all data dependencies go from a child thread to an ancestor thread.

some communication cost.

With a P-processor parallel computer and a scheduling algorithm, given a depth-first multithreaded computation with work T_1, computation depth T_∞, and activation depth $\mathcal{A} = S_1$ possessing average available parallelism $T_1/T_\infty = \Omega(P)$, we would like the scheduling algorithm to compute an execution schedule $\mathcal{X}$ with $T_P(\mathcal{X}) = O(T_1/P)$ and $S_P(\mathcal{X}) = O(S_1 P)$.

In light of the lower-bound, we consider scheduling algorithms for a specific class of depth-first multithreaded computations. In particular, we consider multithreaded computations in which all data dependencies go from a child thread to an ancestor thread as illustrated in Figure 5.1.

Requiring that all data dependencies go from a child thread to an ancestor thread can be viewed as requiring all function invocations (in a functional language) to be strict, and therefore, we refer to this class of computations as *strict multithreaded computations*. For example, many languages express parallelism with the **future** construct [12, 15, 21].

> The expression (**future** X), where X is an arbitrary expression, creates a task to evaluate X and also creates an object known as a *future* to eventually hold the value of X. When created, the future is in an *unresolved*, or *undetermined*, state. When the value of X becomes known, the future *resolves* to that value, effectively mutating into the value of X and losing its identity as a future. Concurrency arises because the expression (**future** X) returns the future as its value without waiting for the future to resolve. Thus, the computation containing (**future** X) can proceed concurrently with the evaluation of X. [21]

Consider the following code fragment:

```
(let ((a (future A))
      (b (future B)))
     (+ C (F a b)))
```

Such a code fragment could appear for example in a Mul-T [21] program. Figure 5.2(a) illustrates the corresponding multithreaded computation. In this example, the thread evaluating this code can spawn child threads to evaluate expressions A and B concurrently; to the parent thread, identifiers a and b are futures until they resolve. Furthermore, evaluation of A and B can proceed concurrently with the parent thread's evaluation of expression C. Once the parent thread has evaluated C (and F) it can go ahead and spawn a child thread to evaluate the invocation (F a b) even if the arguments have not resolved. When a function is invoked with an argument that is a future, the invocation is called *nonstrict*, hence, we call the spawn *nonstrict* as well. To make this computation strict, we must ensure that the function value of F is not invoked until the arguments a and b resolve. In Mul-T, this strictness can be expressed with the **touch** construct as shown in the following code fragment:

```
(let ((a (future A))
      (b (future B)))
     (+ C (F (touch a) (touch b))))
```

In this case, before the parent thread goes to spawn the invocation (F a b), it touches the arguments a and b, thereby forcing the thread to stall until those arguments resolve. Then when it performs the spawn, the arguments are no longer futures, and consequently, the spawn is strict. Figure 5.2(b) illustrates the computation corresponding to this strict version of the code — notice that the data dependencies now conform to the strictness condition. The strict version of this computation still has parallelism: The expressions A, B, and C can still by evaluated concurrently; it's just that evaluation of A and B can no longer operate in parallel with the invocation $(F\ a\ b)$.

Strict computations are also depth-first since requiring all data dependencies to go from a child thread to an ancestor prohibits any data dependency going from one subcomputation of a thread to another subcomputation of that thread.

For strict multithreaded computations, once a thread Γ has been spawned, a single processor can complete the execution of Γ and all of its descendant threads by using a depth-first schedule even if no other progress is made on other parts of the computation. In other words, from the time the thread Γ is spawned until the time Γ terminates, there is always at least one thread from the subtree rooted at Γ that is ready. This property allows us to derive algorithms to schedule the execution of these computations with efficient use of both time and space.

We first show that for any strict multithreaded computation, there exists an execution schedule that achieves linear speedup with linear expansion of space. We demonstrate such schedules by exhibiting a completely synchronous scheduling algorithm that we call GDF (stands for *global depth-first*). On a P-processor parallel computer, for any strict multithreaded computation with work T_1, computation depth T_∞, and activation depth $\mathcal{A} = S_1$ possessing average available parallelism $T_1/T_\infty = \Omega(P)$, algorithm GDF computes a schedule $\mathcal{X}$ such that $T_P(\mathcal{X}) = O(T_1/P)$ and $S_P(\mathcal{X}) = O(S_1 P)$. This algorithm uses a centralized priority queue that is shared by all P processors, hence, the synchronization cost of this algorithm makes it impractical for any reasonably large number of processors.

By modifying GDF we can exhibit an algorithm that is efficient for moderately sized machines. This algorithm, which we call GDF', uses fewer accesses to the global queue while still computing an equally good schedule.

To obtain an algorithm that is efficient for large machines, we use the technique of Karp and Zhang [19] to replace the global priority queue with P local queues, one for each processor. By combining this technique with a new technique to *throttle* the execution and thereby maintain

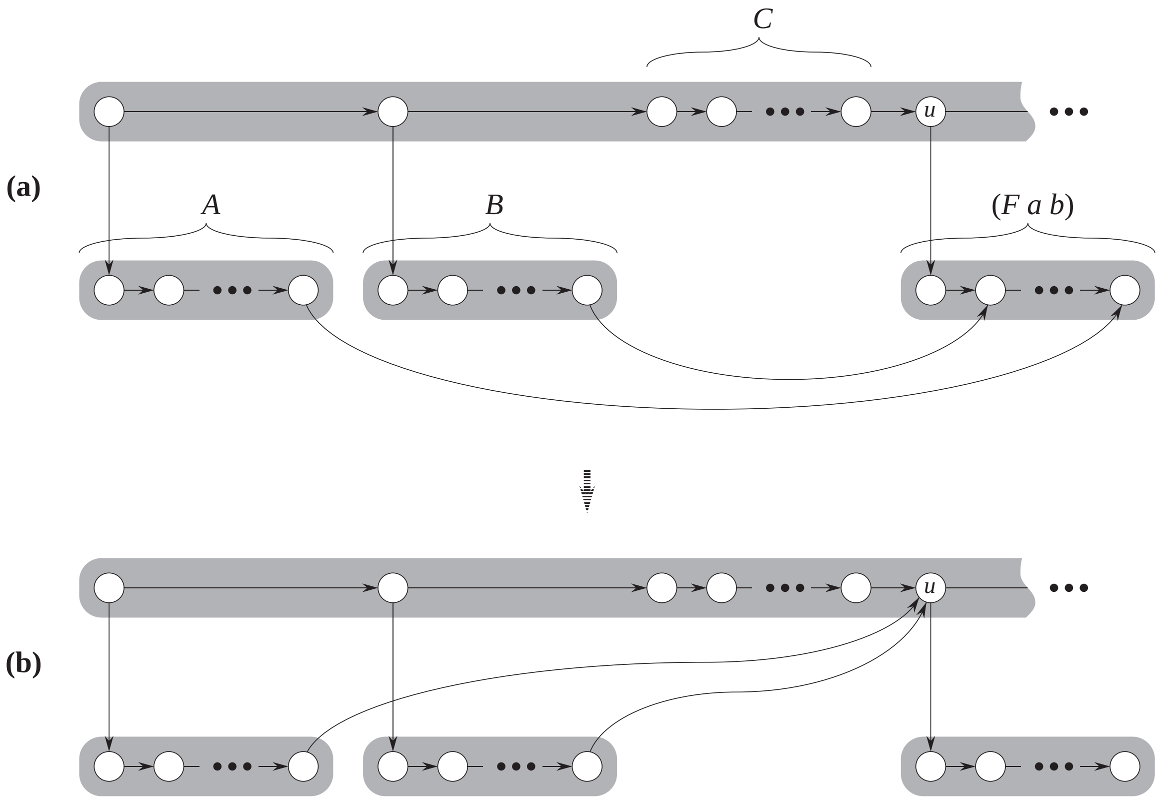

Figure 5.2: **(a)** A nonstrict computation. The parent thread begins by spawning child threads to evaluate expressions A and B. In parallel with the evaluation of A and B, the parent thread can continue on to evaluate expression C. After evaluating C, the parent thread spawns a child thread to evaluate the invocation $(F\ a\ b)$. This spawn can occur even before expression A or B has completed evaluation, in which case at least one of the corresponding identifiers, a or b, is still a future and the spawn is nonstrict. **(b)** A strict version of the same computation. In this case, the parent thread must stall at task u until both expressions A and B have completed evaluation. Thus, the corresponding identifiers, a and b, are no longer futures when the spawn occurs, and the spawn is strict.

a modest degree of synchrony among the processors, we obtain a randomized algorithm that we call LDF (stands for *local depth-first*). For any strict multithreaded computation with $\lg P$ slack in its average available parallelism — that is $T_1/T_\infty = \Omega(P \lg P)$ — algorithm LDF computes a schedule $\mathcal{X}$ with guaranteed space bound $S_P(\mathcal{X}) = O(S_1 P \lg P)$ and expected time bound $\mathrm{E}[T_P(\mathcal{X})] = O(T_1/P)$. This algorithm is simple and distributed (it requires no global control nor any global data structures), and therefore, on a PRAM and certain low-latency, high-bandwidth fixed-connection networks, the scheduling costs are no more than a constant factor of the execution time.

Centralized scheduling algorithms

Algorithm GDF maintains all active threads in a global queue prioritized by activation depth — the deepest threads get highest priority. At each step of the algorithm, the scheduler removes from the queue the P deepest ready threads (if there are fewer than P ready threads, it just removes them all) and assigns them arbitrarily to the P processors so that each processor receives at most one thread. Each processor that has an assigned thread then executes one task from that thread. To complete the step, all surviving threads and all newly spawned threads are placed back into the global queue.

Theorem 5 *For any number P of processors and any strict multithreaded computation with work T_1, computation depth T_∞, and activation depth $\mathcal{A} = S_1$, algorithm* GDF *computes a schedule $\mathcal{X}$ that achieves space $S_P(\mathcal{X}) \leq S_1 P$ and time $T_P(\mathcal{X}) \leq T_1/P + T_\infty$.*

Proof: The time bound follows immediately from Theorem 1 since GDF always produces a greedy schedule.

To prove the space bound, we show that the queue never contains more than P threads (ready or not) that span any activation depth. A thread Γ *spans* an activation depth d, if Γ has activation depth $\mathcal{A}(\Gamma) \geq d$, and either Γ is the root or the parent thread Γ' of Γ has activation depth $\mathcal{A}(\Gamma') < d$. See Figure 5.3. For any time step t during the execution and any activation depth d, let $s(t, d)$ denote the number of active threads that span d at the start of step t. Then the total space $s(t)$ being used at the start of time step t is

$$s(t) = \sum_{d=1}^{S_1} s(t,d). \tag{5.1}$$

By induction on the number of steps, we show that for all t, every activation depth d, has $s(t,d) \leq P$. With this bound, Equation (5.1) shows that $s(t) \leq S_1 P$ for all time t, from which the space bound follows.

The algorithm begins with just one active thread (the root), so for every activation depth d, we have $s(1,d) \leq 1 \leq P$. Now consider any activation depth d, and suppose that for time step t, the induction hypothesis $s(t,d) \leq P$ holds. The computation being strict means that for each of the $s(t,d)$ active threads that span d at the start of step t, there is at least one ready thread with activation depth greater than or equal to d — remember, this property is the crucial property that we get by having all data dependencies go from a child thread to an ancestor thread. Therefore, step t begins with at least $s(t,d)$ ready threads at or deeper than d. The depth-first ordering then ensures that no more than $P - s(t,d)$ threads with depth less than d can execute at step t. Then since the only way to increase the number of threads that span d is to execute a thread shallower than d that spawns a child thread at or deeper than d, step t ends with at most $s(t,d) + (P - s(t,d)) = P$ active threads that span activation depth d. Therefore, $s(t+1,d) \leq P$, and the induction is complete. ■

We can make this algorithm more efficient by reducing the number of accesses to the global queue as follows. The algorithm begins with the root thread assigned to some arbitrary processor and the global queue empty. In general, at the start of a step, some processors have an assigned thread and some don't. Consider a step that begins with n processors that do not have a thread. In this case, to start the step, the scheduler removes from the queue the n deepest ready threads (if there are fewer than n ready threads, it just removes them all) and assigns them arbitrarily to the n processors so that each processor receives at most one thread. Each processor (now considering all P of them) that has an assigned thread then executes one task from that thread. Unless that thread spawns, terminates, or stalls, the processor can keep its

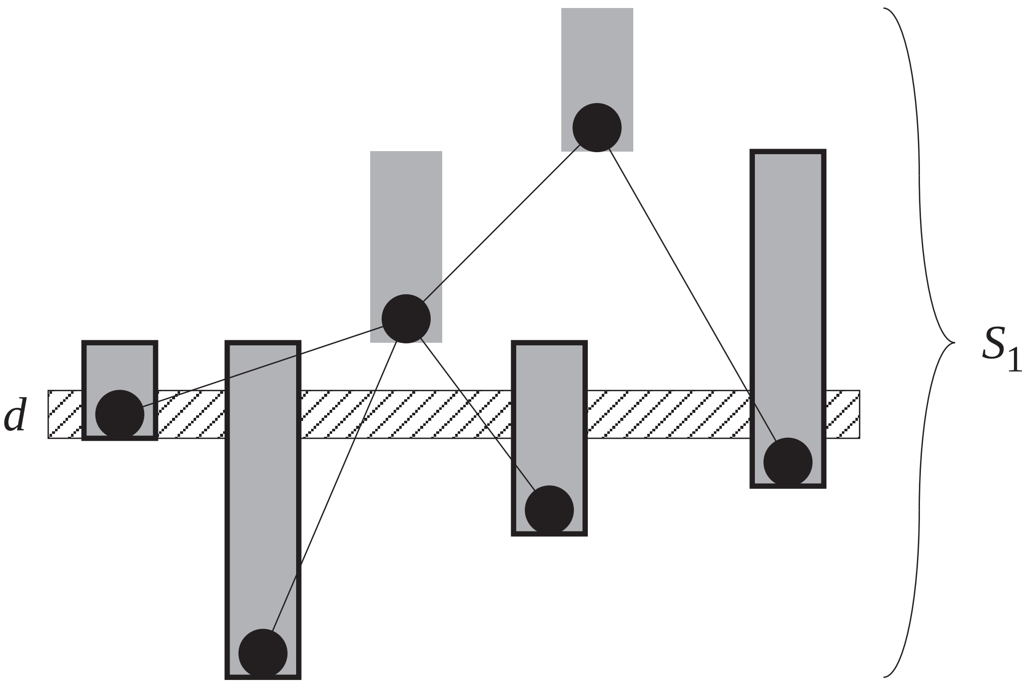

Figure 5.3: The activation tree corresponding to the example computation of Figure 2.1. Each black node corresponds to a thread and the edges correspond to the spawn edges. Associated with each thread is an activation frame depicted by the grey rectangles drawn with height equal to the size of the frame. Notice that the activation frames are located so that the top of a thread's frame is just below the bottom of its parent's frame. In this way each thread's black node is drawn at its activation depth (depth increases in the downward direction). The threads that span activation depth d are indicated by highlighting the activation frame's border.

thread so it will have a thread to start the next step. If the thread stalls, the processor must return it to the global queue, and consequently, the processor will not have a thread to start the next step. Similarly, if the thread terminates, the processor will not have a thread to start the next step. Lastly, if the thread spawns, the processor returns the parent thread (the one it was working on) to the global queue and keeps the child thread, and therefore, in this case, the processor will still have a thread to start the next step.

This version of the algorithm, which we call GDF', achieves the same performance bounds as proved in Theorem 5, but requires access to the global queue only when threads spawn, terminate, or stall.

Theorem 6 *For any number P of processors and any strict multithreaded computation with work T_1, computation depth T_∞, and activation depth $\mathcal{A} = S_1$, algorithm* GDF' *computes a schedule $\mathcal{X}$ that achieves space $S_P(\mathcal{X}) \leq S_1 P$ and time $T_P(\mathcal{X}) \leq T_1/P + T_\infty$.*

Proof: This proof follows the proof of Theorem 5, but we add the following assertion to the induction hypothesis: For any activation depth d, if a step t begins with $s(t,d) \leq P$ active threads that span depth d, then step t begins with no more than $P - s(t,d)$ processors that have a thread with activation depth less than d. If this assertion is true at the start of step t, then at least $s(t,d)$ processors get assigned to threads at or deeper than d, and step $t+1$ begins with $s(t+1,d) \leq P$ active threads spanning d. Also, since no more than $P - s(t,d)$ processors work on threads shallower than d during step t, step $t+1$ begins with no more than $P - s(t,d)$ processors that have a thread shallower than d. We consider two cases based on the relative sizes of $s(t,d)$ and $s(t+1,d)$. If $s(t+1,d) \leq s(t,d)$, then $P - s(t,d) \leq P - s(t+1,d)$, and hence, step $t+1$ begins with no more than $P - s(t+1,d)$ processors that have a thread with activation depth less than d. On the other hand, if $s(t+1,d) > s(t,d)$, then $s(t+1,d) - s(t,d)$ processors must have executed a thread less deep than d that spawned a child thread at or deeper than d during step t. Each of these processors only keeps the thread with depth greater than or equal to d, and consequently, step $t+1$ begins with no more than $P - s(t,d) - (s(t+1,d) - s(t,d)) = P - s(t+1,d)$ processors that have a thread with activation depth less than d. In either case, step $t+1$ begins with no more than $P - s(t+1,d)$ processors having a thread less deep than d, thereby completing the induction. ■

This algorithm may be feasible for a modest number of processors, but for a large number of processors, the cost of synchronization at the global queue becomes prohibitive. To derive a truly scalable and distributed algorithm, we need to split the global queue into P local queues — one for each processor.

Chapter 6

Distributed scheduling algorithms

In a distributed scheduling algorithm, each processor works depth-first out of its own local priority queue. Specifically, to get a thread to work on, a processor removes the deepest ready thread from its local queue. Ideally, we would like the processor to then continue working on that thread until it either stalls, terminates, or spawns, and when the processor does need to enqueue a thread (as in the case when the thread stalls or spawns) or dequeue a new thread, it does so by accessing only its local queue. Of course, this approach could result in processors with empty queues sitting idle while other processors have large queues. Thus, we require each processor to have some access to non-local queues in order to facilitate some type of load balancing.

The technique of Karp and Zhang [19] suggests a randomized algorithm in which threads are located in random queues in order to achieve some balance. At the end of this chapter, we show that the naive adoption of this technique does not work. In order to achieve the desired result, we modify the Karp and Zhang technique by incorporating a new mechanism to enforce a modest degree of synchrony among the processors.

Algorithm LDF operates in iterations with each iteration consisting of a synchronization phase followed by a computation phase and ending with a communication phase. In a synchronization phase, we compute a *cutoff depth* D that is a global value made available to all processors. During the following computation phase, only those threads with activation depth greater than or equal to D can execute. Finally, the communication phase redistributes threads to random locations.

The operation of each phase is governed by a *synchronization parameter* r that affects both the time and space performance of the algorithm. Let LDF(r) denote Algorithm LDF with synchronization parameter r.

In a synchronization phase of LDF(r), we use the synchronization parameter r to compute the cutoff depth D. Each processor p_i, for $i = 1, \ldots, P$, computes the activation depth d_i of its rth deepest ready thread. In other words, d_i is the activation depth for which processor p_i has fewer than r ready threads deeper than d_i but at least r ready threads at or deeper than d_i. Cutoff depth D is then computed simply by

$$D = \max_{1 \leq i \leq P} d_i$$

as illustrated in Figure 6.1.

During the computation phase of LDF(r), each processor executes one task from each ready thread with activation depth greater than or equal to the cutoff depth D in its local queue. We further forbid each processor from executing more than r spawns; if a processor has more than r threads at or deeper than D that want to spawn, it may only execute r of them.

The iteration ends with a communication phase during which each processor must move each ready thread with activation depth greater than or equal to D (as determined at the beginning of the iteration) and each newly spawned thread from its local queue to a queue selected uniformly at random, independently for each thread.

By using the synchronization parameter r to compute the cutoff depth and then ensuring that each processor executes only tasks from threads at or deeper than the cutoff depth while allowing at most r spawns, we get a guaranteed space bound.

Lemma 7 *For any number P of processors and any strict multithreaded computation with activation depth $\mathcal{A} = S_1$, Algorithm* LDF(r) *computes a schedule $\mathcal{X}$ such that $S_P(\mathcal{X}) \leq 2rS_1P$.*

Proof: We show by induction on the number of iterations that no activation depth ever has more than $2rP$ active threads that span it. Specifically, recalling the notation used in the proof

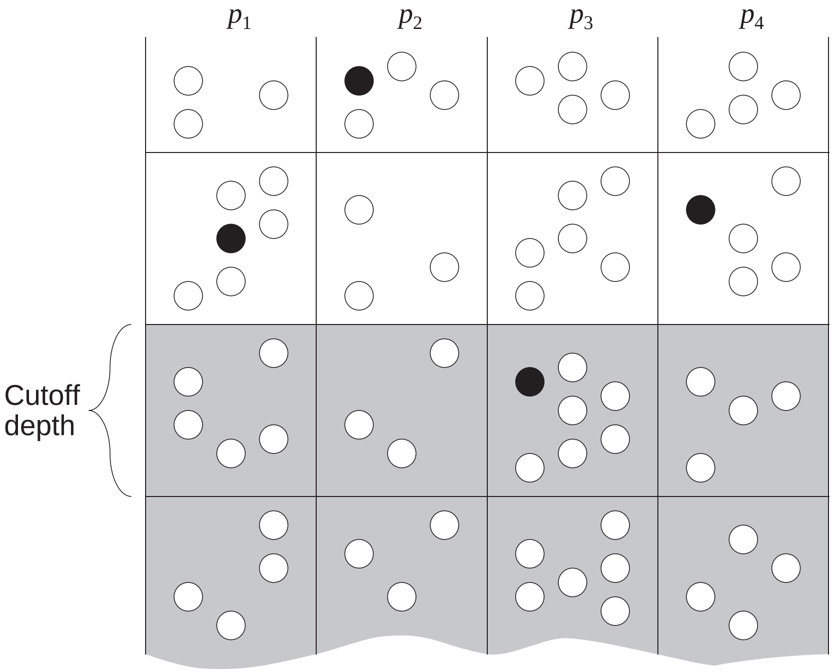

Figure 6.1: Computing the cutoff depth. Each column represents the local priority queue of a processor, and each row represents an activation depth with depth increasing in the downward direction. We depict each thread by a circle located at its activation depth. The ready threads in each queue are ordered by activation depth with ties broken arbitrarily. In this example, the synchronization parameter $r = 12$, and the rth deepest ready thread for each processor is shown in black. The deepest of these black threads determines the cutoff depth. Only the ready threads at or deeper than the cutoff depth — those in the shaded region — can execute during the following computation phase.

of Theorem 5, we show that for every activation depth d and every iteration t of the execution, $s(t,d) \leq 2rP$. The result then follows from Equation (5.1). As before, the base case is obvious.

For any activation depth d and any iteration t of the execution, we consider 2 cases. In the first case, suppose iteration t begins with $rP \leq s(t,d) \leq 2rP$ active threads spanning depth d. Due to the strictness of the computation, there must be at least rP ready threads with activation depth greater than or equal to d, and by pigeon-holing, some processor's local queue must have at least r of them. Therefore, the cutoff depth D will be set with $D \geq d$. Consequently, during the computation phase of iteration t, no thread with activation depth less than d can execute and the iteration ends with no more active threads spanning depth d than it started with. Now suppose iteration t begins with $s(t,d) < rP$ active threads spanning depth d. In this case, during the computation phase, since each processor is only allowed r spawns, the number of active threads that span depth d can increase by at most rP, and therefore, the iteration ends with no more than $2rP$ active threads spanning depth d. In either case, $s(t+1, d) \leq 2rP$, which completes the induction. ■

In order to achieve speedup in the execution time, we must ensure that during the computation phase of each iteration, each processor has some ready threads at or deeper than the cutoff depth. To ensure that the cutoff depth is not set too deep, we must use a large enough synchronization parameter r. On the other hand, the space bound of Lemma 7 is directly proportional to r. By setting $r = 6 \lg P$, the space bound of Lemma 7 becomes $S_P(\mathcal{X}) \leq 12 S_1 P \lg P$, and with high probability, most computation phases take $O(\lg P)$ time and get at least $P \lg P$ tasks executed as we now show.

To analyze the running time, we say that each iteration either *succeeds* or *fails* depending on how many tasks execute. An iteration that begins with at least $P \lg P$ ready threads fails if fewer than $P \lg P$ of the ready threads get a task executed. An iteration that begins with fewer than $P \lg P$ ready threads fails if not all of them get a task executed.

We now show that with the synchronization parameter set to $r = 6 \lg P$, each iteration fails with probability no more than P^{-5}.

Lemma 8 *For any number P of processors, an iteration of Algorithm* LDF$(6 \lg P)$ *fails with probability no more than P^{-5}.*

Proof: Consider an iteration that begins with at least $P \lg P$ ready threads, and suppose that when two threads have the same activation depth, we give each thread a unique identifier to break the tie so we can uniquely identify the $P \lg P$ deepest ready threads. If no local queue contains more than $6 \lg P$ of the $P \lg P$ deepest ready threads, then the synchronization phase sets the cutoff depth so that all $P \lg P$ of these deepest threads are at or deeper than the cutoff depth. Therefore, an iteration that begins with at least $P \lg P$ ready threads succeeds if no local queue contains more than $6 \lg P$ of the $P \lg P$ deepest ready threads.

Consider a particular processor p_i and let the random variable Z_i denote how many of the

$P \lg P$ deepest ready threads start the iteration in the local queue of processor p_i. Each thread is located independently at random, hence, the random variable Z_i has a binomial distribution with $P \lg P$ trials and success probability $1/P$. Therefore,

$$\Pr\{Z_i > 6 \lg P\} \leq \binom{P \lg P}{6 \lg P} \left(\frac{1}{P}\right)^{6 \lg P} .$$

Then from the bound

$$\binom{x}{y} \leq \left(\frac{ex}{y}\right)^y \tag{6.1}$$

and the fact that $6 \geq 2e$, we can upper bound $\Pr\{Z_i > 6 \lg P\}$ by

$$\begin{aligned} \Pr\{Z_i > 6 \lg P\} &\leq \left(\frac{eP \lg P}{6 \lg P}\right) \left(\frac{1}{P}\right)^{6 \lg P} \\ &= \left(\frac{e}{6}\right)^{6 \lg P} \\ &\leq P^{-6}. \end{aligned}$$

Now let $Z = \max_{1 \leq i \leq P} Z_i$. For an iteration that begins with at least $P \lg P$ ready threads, the probability of failure is no more than $\Pr\{Z > 6 \lg P\}$. We can use Boole's Inequality to upper bound $\Pr\{Z > 6 \lg P\}$ by adding the individual probabilities, from which,

$$\Pr\{Z > 6 \lg P\} \leq P \cdot \Pr\{Z_i > 6 \lg P\} \leq P^{-5}.$$

For the case of an iteration that begins with fewer than $P \lg P$ ready threads, the failure probability is still upper bounded by $\Pr\{Z > 6 \lg P\}$ where the random variable Z has the distribution just described. ■

We now show that iterations fail independently of each other. Specifically, we show that knowing whether an iteration t fails provides no information about whether any future iteration fails. The failure of an iteration depends only on how the ready threads are distributed among the processors. Therefore, we need to show that knowing whether iteration t fails provides no information about the distribution of threads at the end of the iteration. Suppose iteration t has cutoff depth D. No matter if iteration t fails or not, the iteration ends with a communication phase in which every ready thread at or deeper than D gets moved to a random location. Thus, iteration t provides no information about the distribution of threads at or deeper than the cutoff depth. Now consider the threads less deep than D. The only part of an iteration that even considers the threads shallower than the cutoff depth is the synchronization phase. Therefore, we need to show that computing the cutoff depth provides no information about the distribution of threads with activation depth less than D. Consider an alternative method for computing the cutoff depth. Let all the processors work in synch from the bottom up. First each processor counts the number of ready threads it has with activation depth S_1. Then each processor adds on the number of ready threads it has with activation depth $S_1 - 1$. We continue in this manner until some processor reaches a count of r (the synchronization parameter). At this depth we stop and set the cutoff depth. In this way the synchronization phase can compute the cutoff depth with the exact same result but without ever considering threads shallower than D. Thus, computing the cutoff depth provides no information about the distribution of threads shallower than the cutoff depth.

With iterations failing independently of each other, we can bound the number of failed iterations, thereby bounding the total number of iterations taken.

Lemma 9 *For any number P of processors and any strict multithreaded computation with work T_1 and computation depth T_∞, for any $\epsilon > 0$, with probability at least $1 - \epsilon$, Algorithm* LDF$(6 \lg P)$ *computes a schedule $\mathcal{X}$ that takes $O(T_1/(P \lg P) + T_\infty + \log_P(1/\epsilon))$ iterations.*

Proof: First we consider the failed iterations. Let the random variable f denote the number of failed iterations. We will show that for any $\epsilon > 0$, the probability that $f \geq eT_1/(P \lg P) + b$ is no more than ϵ when $b = \frac{1}{3}\log_P(1/\epsilon)$. There are at most T_1 iterations since each iteration always results in at least one task being executed, and each iteration fails independently with probability P^{-5}. Therefore, f is bounded by a binomial distribution with T_1 trials and success probability P^{-5}, from which

$$\Pr\left\{f \geq e\frac{T_1}{P\lg P} + b\right\} \leq \binom{T_1}{e\frac{T_1}{P\lg P} + b}\left(\frac{1}{P^5}\right)^{e\frac{T_1}{P\lg P}+b}.$$

Then using Inequality (6.1) we get

$$\begin{aligned}\Pr\left\{f \geq e\frac{T_1}{P\lg P} + b\right\} &\leq \left(\frac{eT_1}{e\frac{T_1}{P\lg P} + b}\cdot\frac{1}{P^5}\right)^{e\frac{T_1}{P\lg P}+b} \\ &\leq \left(\frac{P\lg P}{P^5}\right)^{e\frac{T_1}{P\lg P}+b} \\ &\leq \left(\frac{1}{P^3}\right)^{b} \\ &= P^{-3b},\end{aligned}$$

and $P^{-3b} = \epsilon$ for $b = \frac{1}{3}\log_P(1/\epsilon)$. Thus, with probability at least $1-\epsilon$, $f = O(T_1/(P\lg P) + \log_P(1/\epsilon))$.

Now consider the successful iterations. A successful iteration that begins with at least $P\lg P$ ready threads, executes a task from at least $P\lg P$ of them, and a successful iteration that begins with fewer than $P\lg P$ ready threads, executes a task from every ready thread. Therefore, we can think of each successful iteration as a step in a greedy schedule with $P\lg P$ processors. Then, as in the proof of Theorem 1, we know that there can be no more than $T_1/(P\lg P) + T_\infty$ successful iterations.

Adding together the number of successful iterations and the number of failed iterations completes the proof. ■

Now if we let the random variable X_i denote the time taken by the ith computation phase of Algorithm LDF($6\lg P$), we can give the total time in computation phases as the random variable $X = X_1 + X_2 + \cdots + X_Y$ where Y is the random variable denoting the number of iterations. The time taken by the ith computation phase is proportional to the maximum number of ready threads with activation depth greater than or equal to the cutoff depth in any processor. There can be a total of at most $18P\lg P$ ready threads at or deeper than the cutoff depth — $r = 6P \lg P$ deeper than the cutoff depth and $12P \lg P$ at the cutoff depth (from Lemma 7 with synchronization parameter $r = 6 \lg P$) — and each of these threads is located independently at random. Thus, we can bound each X_i as the size of the largest bin when throwing $18P \lg P$ balls at random into P bins. Furthermore, by the independence argument, the X_i's are independent. We can now bound the random variable X.

Lemma 10 *Let the random variable X denote the sum of Y mutually independent random variables, $X = X_1 + X_2 + \cdots + X_Y$ with each X_i, for $i = 1, \ldots, Y$, distributed as the number of balls in the fullest bin when throwing $P \ln P$ balls independently at random into $P \geq 2$ bins. Then for any $\epsilon > 0$, we have $X = O(Y \ln P + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$.*

Proof: We have

$$\begin{aligned} \Pr\{X \geq aY \ln P + b\} &= \Pr\left\{e^{X/e} \geq e^{(aY \ln P + b)/e}\right\} \\ &\leq \mathrm{E}\left[e^{X/e}\right] e^{-(aY \ln P + b)/e} \end{aligned} \tag{6.2}$$

by Markov's inequality. By the independence of the X_i's,

$$\mathrm{E}\left[e^{X/e}\right] = \prod_{i=1}^{Y} \mathrm{E}\left[e^{X_i/e}\right]. \tag{6.3}$$

From the definition of expectation,

$$\mathrm{E}\left[e^{X_i/e}\right] = \sum_{j=\ln P}^{P \ln P} \Pr\{X_i = j\}\, e^{j/e}.$$

To bound $\mathrm{E}\left[e^{X_i/e}\right]$, we break this sum into pieces. First we break out the terms from $j = \ln P$ to $j = e^3 \ln P - 1$, which yields

$$\mathrm{E}\left[e^{X_i/e}\right] = \sum_{j=\ln P}^{e^3 \ln P - 1} \Pr\{X_i = j\}\, e^{j/e} + \sum_{j=e^3 \ln P}^{P \ln P} \Pr\{X_i = j\}\, e^{j/e}. \tag{6.4}$$

The first of these sums we bound by factoring out the largest term and upper-bounding the sum of probabilities by 1:

$$\begin{aligned}
\sum_{j=\ln P}^{e^3\ln P-1} \Pr\{X_i=j\}\, e^{j/e} &\le \sum_{j=\ln P}^{e^3\ln P-1} \Pr\{X_i=j\}\, e^{e^2\ln P} \\
&= e^{e^2\ln P} \sum_{j=\ln P}^{e^3\ln P-1} \Pr\{X_i=j\} \\
&\le e^{e^2\ln P}. \qquad (6.5)
\end{aligned}$$

To bound the second sum in Equation (6.4), we further break the range of the index variable j into smaller pieces indexed by $k = 3, \ldots, \lceil \ln P \rceil - 1$, with piece k going from $j = e^k \ln P$ to $j = e^{k+1}\ln P - 1$:

$$\begin{aligned}
\sum_{j=e^3\ln P}^{P\ln P} \Pr\{X_i=j\}\, e^{j/e} &= \sum_{k=3}^{\lceil \ln P\rceil -1} \left(\sum_{j=e^k\ln P}^{e^{k+1}\ln P-1} \Pr\{X_i=j\}\, e^{j/e} \right) \\
&\le \sum_{k=3}^{\lceil \ln P\rceil -1} \left(e^{e^k\ln P} \sum_{j=e^k\ln P}^{e^{k+1}\ln P-1} \Pr\{X_i=j\} \right) \\
&\le \sum_{k=3}^{\lceil \ln P\rceil -1} e^{e^k\ln P} \Pr\{X_i \ge e^k \ln P\} \\
&= \sum_{k=3}^{\lceil \ln P\rceil -1} P^{e^k} \Pr\{X_i \ge e^k \ln P\}. \qquad (6.6)
\end{aligned}$$

Now we can bound $\Pr\{X_i \ge e^k \ln P\}$ by the same technique as in Lemma 8, since X_i has the same distribution has the random variable Z considered in the proof of Lemma 8:

$$\begin{aligned}
\Pr\{X_i \ge e^k \ln P\} &\le P \binom{P\ln P}{e^k \ln P} \left(\frac{1}{P}\right)^{e^k \ln P} \\
&\le P e^{-(k-1)e^k \ln P} \\
&= P^{-(k-1)e^k+1}.
\end{aligned}$$

Substituting this bound into Inequality (6.6) yields

$$\begin{aligned}
\sum_{j=e^3\ln P}^{P\ln P} \Pr\{X_i = j\}\, e^{j/e} &\le \sum_{k=3}^{\lceil \ln P\rceil - 1} P^{e^k} P^{-(k-1)e^k+1} \\
&\le \sum_{k=3}^{\infty} P^{-(k-2)e^k+1} \\
&\le 1, \qquad (6.7)
\end{aligned}$$

since the sum is bounded by the geometric sum $\sum_{k=1}^{\infty} 2^{-k} = 1$. Now, we can substitute Inequality (6.5) and Inequality (6.7) back into Equation (6.4), producing

$$\begin{aligned}
\mathrm{E}\left[e^{X_i/e}\right] &\le e^{e^2\ln P} + 1 \\
&\le e^{(e^2+1)\ln P}.
\end{aligned}$$

Substituting this bound into Equation (6.3) and then substituting into Inequality (6.2), we obtain

$$\begin{aligned}
\Pr\{X \ge aY\ln P + b\} &\le e^{((e^2+1)\ln P)Y} e^{-(aY\ln P+b)/e} \\
&= \exp\left(-\left(\frac{a}{e} - e^2 - 1\right) Y \ln P - \frac{b}{e}\right) \\
&\le \exp\left(-\frac{b}{e}\right)
\end{aligned}$$

for $a \ge e^3 + e$. Thus, with $b = e\ln(1/\epsilon)$, we obtain

$$\Pr\left\{X \ge (e^3+e)Y\ln P + e\ln(1/\epsilon)\right\} \le \epsilon.$$

■

We can now characterize the time and space usage for execution schedules computed by the LDF algorithm with synchronization parameter $r = 6\lg P$.

Theorem 11 *For any number $P \geq 2$ of processors and any strict multithreaded computation with work T_1, computation depth T_∞, and activation depth $\mathcal{A} = S_1$, Algorithm* LDF$(6 \lg P)$ *computes a schedule $\mathcal{X}$ that uses space $S_P(\mathcal{X}) = O(S_1 P \lg P)$, and for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the schedule uses time $T_P(\mathcal{X}) = O(T_1/P + T_\infty \lg P + \lg(1/\epsilon))$.*

Proof: The space bound follows directly from Lemma 7 with synchronization parameter $r = 6 \lg P$. The time $T_P(\mathcal{X})$ is the total time taken in computation phases. Let the random variable Y denote the number of iterations. Then we can decompose $T_P(\mathcal{X})$ as a sum of Y mutually independent random variables, $T_P(\mathcal{X}) = X_1 + X_2 + \cdots + X_Y$ with each X_i distributed as the size of the fullest bin when throwing $18P \lg P$ balls independently at random into P bins. Using $\epsilon/2$ as the value of ϵ in Lemma 9, we obtain $Y = O(T_1/(P \lg P) + T_\infty + \log_P(1/\epsilon))$ with probability at least $1 - \epsilon/2$. Then, using $\epsilon/2$ as the value of ϵ in Lemma 10, we obtain $T_P(\mathcal{X}) = O(Y \lg P + \lg(1/\epsilon))$ with probability at least $1 - \epsilon/2$ (using $18P \lg P$ instead of $P \ln P$ only affects the constant). Thus, with probability at least $1 - \epsilon$, the total time taken in computation phases is $T_P(\mathcal{X}) = O(T_1/P + T_\infty \lg P + \lg(1/\epsilon))$. ■

Corollary 12 *For any number $P \geq 2$ of processors and any strict multithreaded computation with work T_1 and computation depth T_∞, Algorithm* LDF$(6 \lg P)$ *computes a schedule $\mathcal{X}$ with expected execution time* $\mathrm{E}\left[T_P(\mathcal{X})\right] = O(T_1/P + T_\infty \lg P)$.

Proof: Just use $\epsilon = 1/P$ in Theorem 11 to get $T_P(\mathcal{X}) = O(T_1/P + T_\infty \lg P)$ with probability at least $1 - 1/P$. Then

$$\begin{aligned} \mathrm{E}\left[T_P(\mathcal{X})\right] &\leq \left(1 - \frac{1}{P}\right) O\left(\frac{T_1}{P} + T_\infty \lg P\right) + \frac{1}{P} T_1 \\ &= O\left(\frac{T_1}{P} + T_\infty \lg P\right) . \end{aligned}$$

■

The LDF$(6 \lg P)$ algorithm achieves linear expected speedup when the computation has average available parallelism $T_1/T_\infty = \Omega(P \lg P)$.

We can view the $\lg P$ factors in the space bound and the average available parallelism required to achieve linear speedup as the computational slack required by Valiant's bulk-synchronous model [33]. The space bound $S_P(\mathcal{X}) = O(S_1 P \lg P)$ indicates that Algorithm LDF($6 \lg P$) requires memory to scale sufficiently to allow each *physical* processor enough space to simulate $\lg P$ *virtual* processors. Given this much space, the time bound $\mathrm{E}\left[T_P(\mathcal{X})\right] = O(T_1/P + T_\infty \lg P)$ then demonstrates linear expected speedup provided the computation has $\lg P$ slack in the average available parallelism.

Practical considerations

In many models of parallel computation, the queueing, synchronization, and communication costs for Algorithm LDF($6 \lg P$) are only a constant fraction of the execution time. If a global max across the P processors can be accomplished in $O(\lg P)$ time, then each synchronization phase takes only $O(\lg P)$ time, and since each computation phase takes $\Omega(\lg P)$ time, the synchronization phases take at most a constant fraction of the total time. To ensure that the communication costs make up only a constant fraction of the total time, each processor must be able to send $w = \Omega(\lg P)$ threads to random processors in $O(w)$ time. For each thread, the communication may involve sending just a word or two of thread description, or it my involve sending the entire activation frame. When the amount of information that needs to be sent with each thread is just some constant amount, then these requirements are met by a hypercube or indirect butterfly using Ranade's algorithm [28] to do the routing.

In order to facilitate the analysis of the LDF algorithm, we had to use a rather large synchronization parameter, but in practice, we expect that Algorithm LDF can be implemented with significantly smaller values of r and a small constant in the expected time bound of Corollary 12. With greater care in the analysis, the synchronization parameter can be reduced from $6 \lg P$ to $4 \lg P$. This reduction in r, reduces the space bound of Theorem 11 from $12 S_1 P \lg P$ to $8 S_1 P \lg P$. The constant hidden in the expected time bound of Corollary 12 then works out to be slightly less than 69; as the number P of processors increases, however, this constant approaches 34. These constants are, of course, artifacts of the analysis. A proper value for the synchronization parameter should be determined empirically. With fairly large machines, values of r much smaller than $4 \lg P$ should work to yield small constants in both the space and expected time bounds.

If implemented, LDF(r) can be modified to allow more asynchrony in the execution, require less thread migration, and take better advantage of specific processor architectures. In particular, during an iteration, each processor can work on threads in any way it desires so long it obeys the following rules.

1. Only threads at or deeper than the cutoff depth may execute.
2. Only r spawns may be performed.
3. Each thread at or deeper than the cutoff depth must finish the iteration at a random location.

With these rules, an iteration can continue for an arbitrarily long time. The computation phase only has to end when a constant fraction of the processors no longer have work to do. In the case of a computation phase in which more than a constant fraction of the processors go idle before $\lg P$ steps, the phase cannot end until each of the other processors has executed at least one task from each of its threads at or deeper than the cutoff depth (modulo rule 2). Once enough processors go idle, the communication phase begins to ensure that each processor observes rule 3 (and this last provision if necessary), and then the iteration ends. During the computation phase, some processors may want to interleave the execution of multiple threads while others may prefer long runs with a single thread.

Bounding individual processor storage requirements

The space bound of Theorem 11 is an aggregate bound, but in a distributed memory machine, we may want to bound the space associated with each individual processor's queue. In the LDF algorithm, each active thread is located in the local queue of a processor chosen at random, so we assume that each activation frame is located in the local memory of the same randomly chosen processor as its associated active thread. Since Lemma 7 shows that the aggregate space used by Algorithm LDF(r) is bounded by $2rS_1P$, we would like some way to ensure that each individual processor requires space bounded by $O(rS_1)$.

Since activation frames are located in randomly chosen processors, we can show that at any given iteration, the expected spaced needed by any given processor is no more than $2rS_1$. Suppose that at some iteration t, there are k active threads with frame sizes $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k$. Consider a particular processor p, and let the random variable W denote the total space being used by activation frames located in the memory of processor p. We can decompose W as the weighted sum of mutually independent indicator random variables:

$$W = \mathcal{F}_1 W_1 + \mathcal{F}_2 W_2 + \cdots + \mathcal{F}_k W_k$$

where the random variable W_i indicates whether the ith active thread is located at processor p. Since each active thread is located at a processor chosen uniformly at random, the expected value of W_i is given by $\mathrm{E}[W_i] = \Pr\{W_i = 1\} = 1/P$. Then we can bound the expected value of W by

$$\begin{aligned}
\mathrm{E}[W] &= \mathcal{F}_1 \mathrm{E}[W_1] + \mathcal{F}_2 \mathrm{E}[W_2] + \cdots + \mathcal{F}_k \mathrm{E}[W_k] \\
&= \frac{1}{P} \sum_{i=1}^{k} \mathcal{F}_i \\
&\leq \frac{1}{P} 2rS_1P \\
&= 2rS_1,
\end{aligned}$$

since the sum of the frame sizes is bounded in Lemma 7 by $2rS_1P$.

To show that for any given iteration t, with high probability, no processor uses more than $O(rS_1)$ space for activation frame storage, we use the following result due to Raghavan [27, Theorem 1].

Lemma 13 (Raghavan) *Let $a_1, a_2, \ldots, a_k$ be reals in $(0,1]$. Let $\psi_1, \psi_2, \ldots, \psi_k$ be independent Bernoulli trials, and let $\Psi = \sum_{i=1}^{k} a_i \psi_i$. Then for any $\delta > 0$,*

$$\Pr\{\Psi > (1+\delta)\mathrm{E}[\Psi]\} \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mathrm{E}[\Psi]} .$$

■

Setting $a_i = \mathcal{F}_i / S_1$, we can bound $\Pr\{W > 2erS_1\}$ by applying Lemma 13 to the random variable Ψ with $\mathrm{E}[\Psi] = 2r$. Then

$$\begin{aligned}
\Pr\{W > 2erS_1\} &\leq \Pr\{\Psi > e\mathrm{E}[\Psi]\} \\
&\leq \left(\frac{e^{e-1}}{e^{e}}\right)^{\mathrm{E}[\Psi]} \\
&= e^{-2r} .
\end{aligned}$$

If the synchronization parameter is set with $r = r' \ln P$ where $r' \geq 1$, this probability is no more than $P^{-2r'}$. Then, since there are P processors, the probability that any processor uses more than $2erS_1 = O(rS_1)$ space at iteration t is bounded by $P^{-(2r'-1)}$.

This probabilistic bound shows that with an appropriate choice of synchronization parameter, we can allow each processor $O(rS_1)$ space and ensure that no processor ever exceeds this space allotment by simply rerandomizing the thread locations any time a processor fills up its allotted space. As we just proved, the probability that rerandomizing is needed at any given iteration is no more than $P^{-(2r'-1)}$. Therefore, the expected number of times that rerandomizing is needed over the course of the entire execution is no more than $T_1 / P^{2r'-1}$. If rerandomizing can be accomplished in $O(rS_1)$ time — as is the case with a fully-connected, hypercube, or indirect butterfly network — then the total expected time taken by rerandomization is no more than

$$O\left(\frac{T_1}{P^{2r'-1}} rS_1\right) = O\left(\frac{T_1}{P^{2r'-1}} r' S_1 \lg P\right) .$$

This total expected rerandomization time is $O(T_1/P)$ provided $r' = \Omega(1 + \log_P S_1)$. Thus, by setting the synchronization parameter to $r = \Theta(\lg P + \lg S_1)$ and rerandomizing thread locations when any processor fills its space allotment, the LDF algorithm achieves linear expected speedup (provided the computation has $\lg P$ slack in its average available parallelism) with each processor's storage requirement bounded by $O(S_1(\lg P + \lg S_1))$. When S_1 is bounded by a polynomial in P, this space bound is $O(S_1 \lg P)$.

Simple strategies that don't work

To conclude this chapter, we now show that some simpler ways of adopting the Karp and Zhang technique do not work.

The most natural thing to try is to have each processor work depth-first out of its local queue and spawn new threads to random locations. Specifically, when a processor executes a task from a thread, if it spawns a new thread, the original thread is kept locally and the new child thread is moved to the queue of a processor selected uniformly at random from all P processors. With this method, once a thread gets spawned and placed into a random queue, it never has to migrate. Unfortunately, this method does not work as the following scenario illustrates. Suppose a processor p has as its deepest thread a thread that just keeps spawning children — a loop with many iterations for example — and each child thread has a unit size activation frame. Suppose also that this loop thread is at activation depth d and all the other processors are busy executing long threads at activation depths greater than $d + 1$. In this case, most of the invocations (which have depth $d + 1$) spawned by the loop thread land in the queues of the other $P - 1$ processors and languish there. The occasional invocation that lands at processor p temporarily interrupts the loop thread, but if each loop invocation is just a short thread, processor p quickly resumes executing the loop thread. Thus, the loop invocations just keep piling up and eventually overflow memory.

To fix this problem, we must force threads to migrate. After a processor executes a task from a thread, it moves that thread to the local queue of a processor selected uniformly at random, and as before, any newly spawned threads are placed at random. Unfortunately, even this method does not work as the following scenario illustrates. Consider an activation depth d

and suppose the threads at d are long sequential threads with unit size activation frames and the threads at depth $d-1$ just keep spawning these long threads. At each step, if a processor has a depth d thread, it just executes a task from that thread and then moves that thread to a random processor; otherwise, it executes a depth $d-1$ thread which spawns a child at depth d. Therefore, if we look at the queues at depth d as bins and the threads as balls, we have the following process. At each step, one ball is removed from each non-empty bin and P new balls are thrown at random into the P bins. If we consider this process over n steps and consider the balls arriving in a particular bin, we have a binomial distribution with mean n and standard deviation $\Theta(\sqrt{n})$. Thus, we can show that the expected number of balls that arrive into the fullest bin is $n+\Omega(\sqrt{n})$. During this time, at most n balls are removed from this bin, hence, this bin contains $\Omega(\sqrt{n})$ balls at the end of these n steps. Recall that each ball corresponds to an activation frame, and therefore, this probabilistic analysis shows that over time, some queue's size grows as the square root of the elapsed time.

The basic problem in the above scenario is that with a purely random process without any global control, over time, processors get out of synch with each other. Even though there may be lots of deep threads in the system, every once in a while, some processor will be without any of these deep threads and therefore will execute a task from a shallow thread that spawns a child. Thus, over time, the number of threads in the system can just keep growing. Our solution to this problem uses a moderate degree of global control to throttle the execution of processors that get out of synch. We implement this throttle by maintaining a cutoff depth to ensure that all processors only execute threads that are among the deepest in the system; a processor that does not have any of these deepest threads cannot execute any tasks until it gets some of these deepest threads.

Chapter 7

Scheduling nonstrict, depth-first multithreaded computations

The algorithms of Chapterss 5 and 6 for strict multithreaded computations can also be used for nonstrict, depth-first computations — just change the computation to make it strict and then execute the strict computation. Transforming a computation to make it strict involves simply adding data dependency edges as illustrated in Figures 5.1 and 5.2; we call this transformation, *strictifying* the computation. This transformation is always valid for depth-first computations. For arbitrary computations, however, there are examples for which strictifying adds data dependency edges that introduce cycles into the computation; for such computations, nonstrict spawns are *required* in any valid execution schedule.

Consider an arbitrary depth-first multithreaded computation with work T_1, computation depth T_∞ and activation depth $\mathcal{A} = S_1$. Strictifying this computation produces a new computation with the same work and activation depth but with a possibly larger computation depth that we denote $T_\infty^{(\mathrm{s})}$. Executing the strict computation on a P-processor computer with algorithm GDF generates an execution schedule $\mathcal{X}$ with $S_P(\mathcal{X}) \leq S_1 P$ and $T_P(\mathcal{X}) \leq T_1/P + T_\infty^{(\mathrm{s})}$. Such a schedule achieves linear speedup provided the strict computation has sufficient average available parallelism; that is, provided $T_1/T_\infty^{(\mathrm{s})} = \Omega(P)$. In general, any of the algorithms of Chapterss 5 and 6 achieve linear speedup (or linear expected speedup) provided the strict computation has average available parallelism sufficiently large relative to P (or $P \lg P$). When $T_\infty^{(\mathrm{s})}$ is much larger than T_∞, however, the strict computation may not have sufficient average available parallelism even though the original (nonstrict) computation does. The fact that a nonstrict computation may have far more parallelism than its strict counterpart is one of the reasons for nonstrictness. Hence, we would like a technique by which a scheduler can exploit at least some of the parallelism offered by nonstrict spawns.

The lower bound of Theorem 4, however, should temper our expectations. The computations demonstrated in the proof of Theorem 4 are all depth-first, but they use extreme amounts of nonstrictness in order to achieve parallelism. As the theorem shows, exploiting this nonstrict parallelism requires potentially unmanageable amounts of storage. Thus, we cannot hope for a technique that achieves parallel speedup from arbitrary uses of nonstrict spawns while still maintaining reasonable space bounds.

In this chapter, we exhibit a technique that allows a scheduler to exploit some of the parallelism available through nonstrict spawns. This technique allows the scheduler to perform some nonstrict spawns while still maintaining space bounds that are within a constant factor of the bounds it obtains for strict computations. Of course, this technique cannot guarantee any speedup from the nonstrict spawns, but it does guarantee execution time that is no greater than the execution time obtained by strictifying and executing the strict computation.

It is important to realize that when space is bounded, the use of nonstrict spawns when executing a computation can actually result in an execution time that is longer than the execution time that results from simply executing the strictified computation. Suppose we could execute the computation as if it were strictified, but at each step, if there is an idle processor and a thread that is stalled (due to the strictness condition) at a task that wants to spawn, we let the processor go ahead and execute that task thereby performing a nonstrict spawn. For example, in executing the computation of Figure 5.2(a), if at some time step t, execution of the parent thread is at the task u that spawns the invocation $(F\ a\ b)$ and execution of either the child thread evaluating expression A or the child thread evaluating expression B is not complete, then task u can only execute if a processor would otherwise go idle. Performing the spawn requires allocating an activation frame, and this is where the trouble lies as the following scenario illustrates: Suppose there is a single thread Γ computing a value A that is used by lots of other threads. At step t, one processor executes a task from Γ, and instead of idling, some of the other processors perform nonstrict spawns — invoking functions that have A as an argument, for example. At the next step, the same thing happens, and this continues for awhile. Over time, memory gets filled up with the activation frames of these threads that were spawned nonstrictly. To avoid overflowing memory bounds, eventually these nonstrict spawns must cease. At this point, thread Γ is still computing A, and lots of other threads are stalled waiting for A. Now, if Γ wants to spawn a bunch of child threads to help it compute A, it cannot do so since memory is already full. In this case, the nonstrict spawns do not really add any useful parallelism since the spawned threads just stall waiting for A. Useful parallelism could have come from the evaluation of A, but with memory full, that parallelism cannot be exploited. Thus, performing nonstrict spawns may increase processor utilization for a brief spell but at the cost of forcing very low processor utilization for a potentially very long period of time — a period of time that could have had very high processor utilization had those nonstrict spawns not been performed.

To keep the nonstrict spawns from hindering the progress of other parts of the computation, we classify each active thread as either strict or nonstrict and then ensure that the nonstrict threads do not fill up too much memory. When a thread is spawned nonstrictly, we say that the thread itself is nonstrict. A nonstrict thread remains nonstrict until those data dependencies that caused the spawn to be nonstrict in the first place get resolved. Once those data dependencies get resolved, the thread is strict. For example, in executing the computation in Figure 5.2(a), if the child thread that evaluates the invocation $(F\ a\ b)$ is spawned nonstrictly, then that thread remains nonstrict until both the thread evaluating expression A and the thread evaluating expression B terminate thereby resolving the associated data dependencies. A strictly spawned thread is considered strict and remains strict. Observe that from the time an active thread Γ becomes strict until the time Γ terminates, there is always at least one thread from the subtree rooted at Γ that is ready. This crucial property of strict threads in combination with an enforced bound on the space used by nonstrict threads forms the basis for a technique that we call *α-sequestering*.

To ensure that the activation frames of nonstrict threads do not interfere with the progress of strict threads, the α-sequestering technique allocates separate space — the amount is determined by the value of α — for use by the nonstrict threads. By maintaining a separate region of memory for the activation frames of nonstrict threads, α-sequestering allows nonstrictness without adversely affecting the running time. We execute the computation as if it were strictified, but at each step, if there are idle processors and threads that are stalled (due to the strictness condition) at tasks that want to spawn, we allow processors to perform these nonstrict spawns so long as the activation frames of the resulting nonstrict threads do not overflow their region of memory.

We illustrate the effectiveness of α-sequestering in conjunction with the global depth-first algorithm GDF. Suppose we allow nonstrict spawns only so long as no activation depth ever has more than αP active, nonstrict threads that span it. We are not specifying any specific way of prioritizing among nonstrict spawns — we are only saying that nonstrict spawns can only occur when processors would otherwise go idle, and they can only occur so long as no activation depth ever has more than αP active, nonstrict threads that span it. For this reason, we refer to this scheduling policy as the α-sequestered GDF method (rather than algorithm).

Theorem 14 *For any number P of processors and any depth-first multithreaded computation with work T_1, strict computation depth $T_\infty^{(s)}$, and activation depth $\mathcal{A} = S_1$, the α-sequestered* GDF *method computes a schedule $\mathcal{X}$ such that $T_P(\mathcal{X}) \leq T_1/P + T_\infty^{(s)}$ and $S_P(\mathcal{X}) \leq (1+\alpha)S_1 P$.*

Proof: The time bound follows from Theorem 1 since the schedule $\mathcal{X}$ is greedy with respect to the strictified version of the computation.

To prove the space bound, we show that no activation depth ever has more than $(1+\alpha)P$ active threads that span it. Specifically, using the notation from the proof of Theorem 5, we show that for every activation depth d and every time step t, the bound $s(t,d) \leq (1+\alpha)P$ holds. The space bound then follows from Equation (5.1). As before, we prove this bound by induction on the number of time steps, and again, the base case is obvious.

Now, consider a time step t that begins with $s(t,d) \leq (1+\alpha)P$ active threads spanning d. Further, let $s'(t,d)$ denote the number of these threads that are strict. With $s'(t,d)$ active, strict threads spanning d, there must be at least $s'(t,d)$ ready threads at or deeper than d. We consider two cases. In the first case, $s'(t,d) \geq P$. In this case, there are at least P ready threads at or deeper than d, hence, no threads less deep than d execute at step t. Therefore, the number of active threads that span d cannot increase during step t, so $s(t+1,d) \leq s(t,d) \leq (1+\alpha)P$. In the other case, $s'(t,d) < P$, so as many as $P - s'(t,d)$ threads less deep than d may execute during step t. Consequently, the number of threads that span d may increase by as many as $P - s'(t,d)$ but not more. Thus,

$$\begin{aligned} s(t+1,d) &\leq s(t,d) + (P - s'(t,d)) \\ &= P + (s(t,d) - s'(t,d)) \\ &\leq P + \alpha P \end{aligned}$$

since $s(t,d) - s'(t,d)$ is the number of active, nonstrict threads that span d, and this number, by force of the method, is no more than αP. In both cases, $s(t+1,d) \leq (1+\alpha)P$, and the induction is complete. ■

Exactly as with GDF, we can use the α-sequestering technique with algorithm GDF' to yield the α-sequestered GDF' method.

Theorem 15 *For any number P of processors and any depth-first multithreaded computation with work T_1, strict computation depth $T_\infty^{(\mathrm{s})}$, and activation depth $\mathcal{A} = S_1$, the α-sequestered* GDF' *method computes a schedule $\mathcal{X}$ such that $T_P(\mathcal{X}) \leq T_1/P + T_\infty^{(\mathrm{s})}$ and $S_P(\mathcal{X}) \leq (1+\alpha)S_1 P$.*

Proof: This proof follows the proof of Theorem 14, but we add the following assertion to the induction hypothesis: For any activation depth d and time step t, if t begins with $s'(t,d)$ active, strict threads that span d, then t also begins with no more than $\max(P - s'(t,d), 0)$ processors having a thread with activation depth less than d. Proving that this additional assertion holds follows the proof of Theorem 6. ■

This α-sequestering technique can also be used with the local depth-first algorithm LDF. At each iteration, only those threads (strict or nonstrict) at or deeper than the cutoff depth can execute, and each processor is allowed no more than r spawns (strict or nonstrict), where r is the synchronization parameter. Nonstrict spawns are allowed only when a processor would otherwise go idle and only so long as no activation depth ever has more than αP active, nonstrict threads that span it. This α-sequestered LDF method achieves execution time as stated in Theorem 11 but with T_∞ replaced by $T_\infty^{(\mathrm{s})}$; this result follows by making the obvious change in the proof of Lemma 9. The space bound is captured in the following theorem.

Theorem 16 *For any number P of processors and any depth-first multithreaded computation with activation depth $\mathcal{A} = S_1$, the α-sequestered* LDF(r) *method computes a schedule $\mathcal{X}$ such that $S_P(\mathcal{X}) \leq (2r+\alpha)S_1 P$.*

Proof: We show that for any activation depth d and any iteration t, the bound $s(t,d) \leq (2r+\alpha)P$ holds. Again, we prove this bound by induction on the number of iterations, and the base case is obvious.

Now, consider an iteration t that begins with $s(t,d) \leq (2r+\alpha)P$ active threads that span d. And as before, let $s'(t,d)$ denote the number of active, strict threads that span d at the start of iteration t. Consider two cases. In the first case, $s'(t,d) \geq rP$. In this case there are at least rP ready threads at or deeper than d and by pigeon-holing, some processor must have at least r of them. Therefore, the synchronization phase sets the cutoff depth D with $D \geq d$, hence, no thread less deep than d executes at iteration t. Consequently, $s(t+1,d) \leq s(t,d) \leq (2r+\alpha)P$. In the other case, $s'(t,d) < rP$. In this case, the number of active threads that span d may increase but not by more than rP since no processor may execute more than r spawns during an iteration. Then

$$\begin{aligned}
s(t+1,d) &\leq s(t,d) + rP \\
&= (s(t,d) - s'(t,d)) + (s'(t,d) + rP) \\
&\leq (s(t,d) - s'(t,d)) + 2rP \\
&\leq \alpha P + 2rP
\end{aligned}$$

since $s(t,d) - s'(t,d)$ is the number of active, nonstrict threads that span d, and this number, by force of the method, is no more than αP. In both cases, $s(t+1, d) \leq (2r + \alpha)P$, and the induction is complete. ■

By adjusting the value of α, the α-sequestering technique provides some control over the space bounds and the allowable nonstrictness. With $\alpha = 0$, the computation is forced to execute strictly. At the other extreme, with $\alpha = \infty$, the computation may execute with arbitrary amounts of nonstrictness (and achieve execution time within a factor of two of optimal by using a greedy schedule) but with a potentially huge demand on space. In order to maintain space bounds that are within a constant factor of those obtained with strict computations, the value α needs to be no more than a constant (for GDF or GDF') or proportional to the synchronization parameter (for LDF).

The α-sequestering technique does not specify how to schedule nonstrict spawns, it does not specify how to determine whether a particular spawn will be nonstrict, and it does not specify how to keep track of the space being used by nonstrict threads. All of these further specification are needed for a real algorithm or implementation. Furthermore, α-sequestering does not guarantee any speedup from the nonstrict parallelism. Nevertheless, with proper linguistic and runtime mechanisms, α-sequestering may prove feasible, and with new ways to prioritize the nonstrict spawns, α-sequestering may be able to exploit nonstrict parallelism with small values of α and provable speedup for specific uses of nonstrictness in depth-first computations.

Chapter 8

Related work

Storage management for multithreaded computations has been a concern for a number of years. In 1985, Halstead [12] described this problem.

> A classical difficulty for concurrent architectures occurs when there is too *much* parallelism in the program being executed. A program that unfolds into a very large number of parallel tasks may reach a deadlocked state where every task, to make progress, requires additional storage (e.g., to make yet more tasks), and no more storage is available. This can happen even though a sequential version of the same program requires very little storage. In effect, the sequential version executes the tasks one after another, allowing the same storage pool to be reused. By trying to execute all tasks at the same time, the parallel machine may run out of storage.

Nevertheless, precious little prior work has addressed this problem. To date, most existing techniques for controlling storage requirements have consisted of heuristics to either bound storage use by explicitly controlling storage as a resource or reduce storage use by modifying the scheduler's behavior. We are aware of no prior scheduling algorithms with proven time and space bounds.

The storage management problem, as described by Halstead, can be quite pronounced under the execution of a *fair* scheduler. By executing threads in round-robin fashion, a fair scheduler gives each ready thread a fair portion of the execution time. A fair scheduler aggressively exposes parallelism, often resulting in excessive space requirements. Consider the multithreaded computation of Figure 8.1. Let N denote the number of leaf threads (this computation performs a divide-and-conquer algorithm on an input of size N), and suppose each activation frame has unit size. This computation has work $T_1 = \Theta(N)$ and activation depth $\mathcal{A} = \Theta(\lg N)$. Notice also that this computation is depth-first (and strict), and therefore it can be sequentially executed

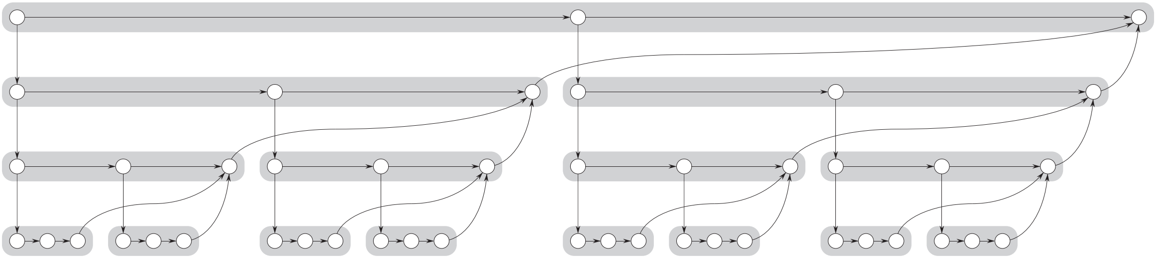

Figure 8.1: A multithreaded computation to perform a divide-and-conquer algorithm. Each non-leaf thread spawns two children. Each child computes a value that it passes back to its parent. Once the parent gets a value back from each child, it computes a result value that it then passes up to its parent.

using space $S_1 = \mathcal{A} = \Theta(\lg N)$. A parallel execution with a fair scheduler, however, executes this computation in (nearly) breadth-first order; at some point in the execution, nearly every leaf thread is active, and therefore, the fair schedule $\mathcal{X}$ (with any number $P \geq 2$ of processors) uses space $S_P(\mathcal{X}) = \Theta(N)$ — an exponential blowup in storage requirements.

In order to curb the excessive exposition of parallelism, and consequent excessive use of space, exhibited by fair scheduling, researchers from the dataflow community have developed heuristics to explicitly manage storage as a resource. The effectiveness of these heuristics is documented with encouraging empirical evidence but no provable time bounds. We consider two of these heuristic techniques: *bounded loops* and the *course-grain throttle*.

Culler's *bounded loops* technique [6, 7, 8] uses compile-time analysis to augment the program code with *resource management code*. For each loop of the program, the resource management code computes a value called the *k-bound*; a k-bounded loop can have at most k iterations simultaneously active. The k-bound represents k *tickets* each of which buys the use of some storage. Once the loop has spawned k iterations, it must wait until one of those iterations completes and relinquishes its ticket; then the loop can use that ticket to spawn another iteration. The compile-time analysis that generates the code that computes the k-bounds is based on heuristics developed from a systematic study of loops in scientific dataflow programs (programs employing only iteration and primitive recursion) [7]. These heuristics attempt to set the k-bounds so that the exposed parallelism is maximized under the constraint that space usage stays within the machine's capacity.

Ruggiero's *course-grain throttle* technique [30] makes storage allocation decisions based on overall machine activity at run-time. When a process (thread) wants to spawn a child, it must request an activation name from the *resource management system.* When the overall level of activity in the machine is high, the resource manager defers these requests, thereby *suspending* the requesting processes. When the activity level falls below a certain threshold, the resource manager begins granting deferred requests giving priority to the lowest, leftmost suspended processes in the process (activation) tree. Like the bounded loops technique, the goal of the coarse-grain throttle is to maximize the exposed parallelism under a fixed space usage constraint.

In contrast with these heuristic techniques, we have chosen to develop an algorithmic foundation that manages storage by allowing programmers to leverage their knowledge of storage requirements for sequentially executed programs. The two techniques just described view storage as a resource that requires explicit management, and they actually modify execution behavior based on these management policies. Such techniques, however, generally have not been needed for programs running on serial machines — when the machine runs out of memory, the program terminates. On most uniprocessor systems, the job of ensuring that the program does not use too much memory rests solely with the programmer, and such systems work because programmers understand the storage model and they understand the execution schedule that orders the invocations of the program's procedures. On parallel systems, however, the storage model is somewhat more complex and predicting the execution order is somewhat more difficult. Nevertheless, this increased complexity does not require encumbering parallel machines with responsibility for bounding storage requirements. Programmers should still be able to understand the storage model, and by developing an algorithmic understanding of scheduling that relates parallel storage requirements to serial storage requirements, programmers should still be able to predict how much storage their programs will use when run on a parallel computer.

Other researchers have also addressed the storage issue by attempting to relate parallel storage requirements to serial storage requirements. Halstead, in completing the quoted paragraph above, made the following observation:

> Ideally, parallel tasks should be created until the processing power of the parallel machine is fully utilized (we may call this *saturation*) and then execution within each task should become sequential. [12]

To emulate this ideal behavior, Halstead considered an *unfair* scheduling policy. When a processor executes a thread that spawns a child, the processor places the parent thread into a local LIFO pending queue and begins work on the child thread. If all the processors remain busy, the parent thread stays in the local pending queue until the child thread terminates. (This execution is exactly the type of depth-first sequential execution that is so familiar to programmers.) If, however, another processor goes idle in the meantime, then it may steal the pending parent thread. Thus, so long as all the processors remain busy, each processor operates depth-first out of its local queue and each local queue's size is bounded by the maximum stack depth in a serial execution. On the strict computation of Figure 8.1, for example, this unfair scheduling policy computes a P-processor execution schedule $\mathcal{X}$ with $S_P(\mathcal{X}) \leq S_1 P$. When we consider more complex computations, even if we just consider strict computations, however, this unfair scheduling policy may exhibit greater than linear space expansion, and in general, predicting or bounding space usage is quite difficult.

Characterizing the performance of Halstead's unfair scheduling policy is even more difficult when we consider time bounds. Though this policy attempts to compute a greedy schedule by allowing idle processors to steal pending threads from other processors, success depends on the thread stealing algorithm. Other researchers [17, 23, 34] have considered variants of unfair scheduling, but none have fully developed or analyzed thread stealing algorithms.

A multithreaded computation with no data dependency edges is equivalent to a backtrack search problem, and in this context, Zhang [36] actually did develop and analyze a thread stealing algorithm. Zhang showed that in a fully connected processor model with P processors, if idle processors choose other processors at random to steal work from, then a binary tree of size N and height h can be search in $O(N/P + h)$ time with high probability. In the context of multithreaded computations with no data dependency edges, this bound translates into a schedule $\mathcal{X}$ that with high probability achieves $T_P(\mathcal{X}) = O(T_1/P + T_\infty)$. Though Zhang did not make the observation, his algorithm also demonstrates linear expansion of space: $S_P(\mathcal{X}) \leq S_1 P$. Other researchers [18, 29] have considered backtrack search on fixed-connection networks, but their algorithms explore the tree in breadth-first order and consequently demonstrate poor space performance.

Chapter 9

Conclusions

The results of this thesis just begin to develop our algorithmic understanding of nonstrictness in multithreaded computations. We have formalized a model of multithreaded computations and developed a working definition to characterize efficient execution schedules with respect to time and space usage. In general, it appears that arbitrary uses of nonstrictness can make efficient parallel execution difficult. In fact, we have demonstrated uses of nonstrictness that make efficient parallel execution provably impossible. This difficulty stands in sharp contrast to the situation with strict computations. For strict computations, we have shown the existence of efficient execution schedules for any number of processors, and further, we have exhibited (fairly) efficient online and distributed algorithms to compute such schedules. Between these extremes, we have a technique that allows the use of some nonstrictness in an otherwise strict computation without degrading the efficiency, but this technique does not guarantee any benefit from the nonstrictness.

Even among the strict computations, some open problems still remain, most notably with respect to efficient and practical scheduling algorithms. For one thing, none of the algorithms presented in this thesis deal with the space used by persistent data structures. Also, the LDF algorithm of Chapter 6 does not take any advantage of locality. An algorithm that can keep groups of closely related threads in the same processor or that can exploit specific fixed-connection networks to keep related threads close to one another would alleviate some of the communication costs. The work on lazy task creation [23] and the work on dynamic tree embedding [3, 22] may provide some pointers in this direction. Of course, an algorithm that removes the $\lg P$ factor from the space bound of LDF would be a nice improvement.

Other algorithmic improvements to LDF might include: an algorithm that performs less thread migration, a technique to keep track of thread location when threads do migrate, a more asynchronous algorithm, and an incremental rebalancing technique to keep the individual queues bounded. Finally, it would be interesting to see if a deterministic distributed algorithm is possible.

Turning back to nonstrict computations, we find a vast range of uncharted territory. Currently, α-sequestering is the only technique we know of that allows nonstrictness in the execution of multithreaded computations while maintaining reasonable space and time bounds. This technique may be practical if efficient support mechanisms can be developed. In this case, with simple algorithms for scheduling the nonstrict spawns, the α-sequestered methods described in Chapter 7 may perform well in practice using small values of α.

We believe, however, that deriving any real benefit from either α-sequestering or any other technique for executing nonstrict computations depends on developing a fundamental understanding of how nonstrictness can be used to realize increased parallelism. Computations that are inherently highly parallel can be packaged into programs in such a way that the parallelism can only be exploited through such extensive use of nonstrictness that efficient execution on a parallel computer is impossible. Therefore, we need to understand how to write programs in such a way that nonstrict parallelism can be exploited. Developing such an understanding might involve identifying useful patterns of usage for nonstrictness and developing algorithms to schedule computations that follow these patterns. Such advances would greatly increase the utility of nonstrictness and in general would expand the class of multithreaded computations for which efficient methods of execution are known.

Bibliography

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 1990. Also: MIT Laboratory for Computer Science Technical Report MIT/LCS/TM-450.

[2] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *Computer*, 21(8):9–24, August 1988.

[3] 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.

[4] Bob Boothe and Abhiram Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 214–223, Gold Coast, Australia, May 1992.

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

[6] 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. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-332.

[7] 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. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-446.

[8] 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.

[9] 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.

[10] 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 Lab Memo MIT/AI/TR-1069.

[11] V. G. Grafe and J. E. Hoch. The Epsilon-2 hybrid dataflows architecture. In *COMPCON 90*, pages 88–93, San Francisco, California, February 1990.

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

[13] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, Honolulu, Hawaii, May 1988.

[14] Waldemar Horwat. Concurrent Smalltalk on the message-driven processor. Technical Report MIT/AI/TR-1321, MIT Artificial Intelligence Laboratory, September 1991.

[15] 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.

[16] Robert A. Iannucci. Toward a dataflow / von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, Honolulu, Hawaii, May 1988. Also: MIT Laboratory for Computer Science, Computation Structures Group Memo 275.

[17] 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.

[18] 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.

[19] Richard M. Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 290–300, Chicago, Illinois, May 1988.

[20] Stephen W. Keckler and William J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213, Gold Coast, Australia, May 1992.

[21] 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.

[22] 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.

[23] 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. Also: MIT Laboratory for Computer Science Technical Report MIT/LCS/TM-449.

[24] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, Jerusalem, Israel, May 1989. Also: MIT Laboratory for Computer Science, Computation Structures Group Memo 292.

[25] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 1992. Also: MIT Laboratory for Computer Science, Computation Structures Group Memo 325–1.

[26] Gregory M. Papadopoulos and Denneth R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 342–351, Toronto, Canada, May 1991. Also: MIT Laboratory for Computer Science, Computation Structures Group Memo 330.

[27] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37(2):130–143, October 1988.

[28] Abhiram Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 185–194, Los Angeles, California, October 1987.

[29] 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.

[30] 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.

[31] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, Yoshinori Yamaguchi, and Yasuhito Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 146–155, Gold Coast, Australia, May 1992.

[32] Kenneth R. Traub. *Sequential Implementation of Lenient Programming Languages.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1988. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-417.

[33] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

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

[35] 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.

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