SIAM J. COMPUT.
Vol. 27, No. 1, pp. 202–229, February 1998

009

SPACE-EFFICIENT SCHEDULING OF MULTITHREADED COMPUTATIONS*

ROBERT D. BLUMOFE† AND CHARLES E. LEISERSON‡

Abstract. This paper considers the problem of scheduling dynamic parallel computations to achieve linear speedup without using significantly more space per processor than that required for a single-processor execution. Utilizing a new graph-theoretic model of multithreaded computation, execution efficiency is quantified by three important measures: T_1 is the time required for executing the computation on a 1 processor, T_∞ is the time required by an infinite number of processors, and S_1 is the space required to execute the computation on a 1 processor. A computation executed on P processors is time-efficient if the time is $O(T_1/P + T_\infty)$, that is, it achieves linear speedup when $P = O(T_1/T_\infty)$, and it is space-efficient if it uses $O(S_1 P)$ total space, that is, the space per processor is within a constant factor of that required for a 1-processor execution.

The first result derived from this model shows that there exist multithreaded computations such that no execution schedule can simultaneously achieve efficient time and efficient space. But by restricting attention to "strict" computations—those in which all arguments to a procedure must be available before the procedure can be invoked—much more positive results are obtainable. Specifically, for any strict multithreaded computation, a simple online algorithm can compute a schedule that is both time-efficient and space-efficient. Unfortunately, because the algorithm uses a global queue, the overhead of computing the schedule can be substantial. This problem is overcome by a decentralized algorithm that can compute and execute a P-processor schedule online in expected time $O(T_1/P + T_\infty \lg P)$ and worst-case space $O(S_1 P \lg P)$, including overhead costs.

Key words. parallel computing, multithreaded computing, parallel algorithms, scheduling algorithms, randomized algorithms, strict execution, stack memory

AMS subject classifications. 68Q22, 68Q25, 68M20

PII. S0097539793259471

1. Introduction.

In the course of investigating schemes for general-purpose MIMD-style parallel computation, many diverse research groups have agreed on multithreading as a dominant paradigm. As an example, modern dataflow systems [16, 19, 25, 33, 34, 35, 40, 41] 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 dynamically orders execution of the threads. Other systems employ schedulers that dynamically order threads based on the availability of data in shared-memory multiprocessors [1, 10, 23] or message arrivals in message-passing multicomputers [2, 17, 29, 44].

Rapid execution of a multithreaded computation on a parallel computer requires exposing and exploiting parallelism in the computation by keeping enough threads concurrently alive to keep the processors of the computer busy. If processors are busy most of the time, the P-processor execution schedule $\mathcal{X}$ of the computation exhibits

*Received by the editors December 9, 1993; accepted for publication (in revised form) January 12, 1996. This research was supported in part by the Defense Advanced Research Projects Agency under grant N00014-91-J-1698. An extended abstract of this paper appeared in the *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing (STOC)*, San Diego, CA, ACM, New York, 1993, pp. 362–371.

http://www.siam.org/journals/sicomp/27-1/25947.html

†Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188 (rdb@cs.utexas.edu). This research was conducted at the MIT Laboratory for Computer Science with additional support from a National Science Foundation Graduate Fellowship.

‡MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139 (cel@mit.edu).

linear speedup: the running time $T(\mathcal{X})$ is order P times faster than the optimal running time T_1 with 1 processor, that is, $T(\mathcal{X}) = O(T_1/P)$.

In attempting to expose parallelism, however, schedulers often end up exposing more parallelism than the computer can actually exploit, and since each living thread requires the use of a certain amount of memory, such schedulers can easily overrun the memory capacity of the machine [15, 22, 24, 39, 43]. To date, the space requirements of multithreaded computations have been managed with heuristics or not at all [14, 15, 22, 24, 26, 32, 39, 43]. In this paper, we use algorithmic techniques to address the problem of managing storage for multithreaded 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 total space $S(\mathcal{X})$ required by a P-processor execution schedule $\mathcal{X}$ 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 P-processor execution schedule $\mathcal{X}$ exhibits at most linear expansion of space, that is, $S(\mathcal{X}) = O(S_1 P)$.

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 a factor of ρ speedup, that is, execution time $T(\mathcal{X}) \leq T_1/\rho$, must use space at least $S(\mathcal{X}) \geq (1/4)(\rho - 1)\sqrt{T_1} + S_1$. For such a computation, even achieving a factor of 2 speedup ($\rho = 2$) requires space that grows as a function 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, although the parameters may be evaluated in parallel. Computations such as parallel divide-and-conquer, backtrack search, branch-and-bound, and game-tree search are all strict.

We show that for any strict multithreaded computation and any number P of processors, there exists a P-processor execution schedule $\mathcal{X}$ that achieves time $T(\mathcal{X}) \leq T_1/P + T_\infty$, where T_∞ is the optimal execution time on an infinite number of processors, and space $S(\mathcal{X}) \leq S_1 P$. Such a schedule exhibits linear expansion of space and linear speedup, $T(\mathcal{X}) = O(T_1/P)$, 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 prove such schedules exist 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 $T_1/T_\infty = \Omega(P \lg P)$.

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 paper is organized as follows. Section 2 develops a formal model of multithreaded computation and execution schedules. In section 3, we characterize multithreaded computations with three parameters and state some basic bounds relating these parameters to execution time and space. The lower bound for general multithreaded computations is presented in section 4. The upper bound

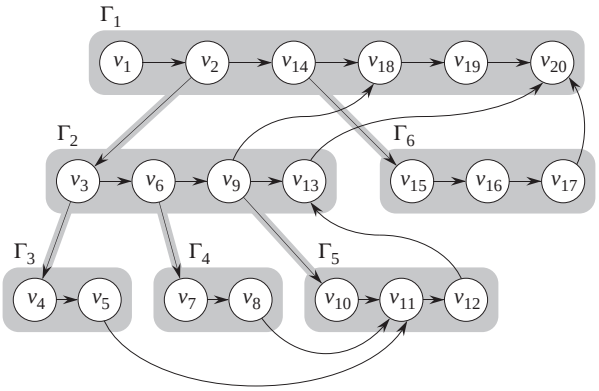


FIG. 2.1. *A multithreaded computation. This computation contains* 20 *tasks* $v_1, v_2, \ldots, v_{20}$ *and six threads* $\Gamma_1, \Gamma_2, \ldots, \Gamma_6$.

for strict computations and the technique for handling limited nonstrictness are presented in section 5. Section 6 presents a distributed scheduling algorithm for strict computations. Finally, in section 7 we conclude with a discussion of related and future work.

2. A model for multithreaded computation. This section defines the model of multithreaded computation that we use in this paper. 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. Thread Γ_5 of this example contains three tasks: v_{10}, v_{11}, and v_{12}. 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.

A P-processor *execution schedule* for a multithreaded computation determines which processors of a P-processor parallel computer execute which tasks at each step. In any given step of an execution schedule, each processor either executes a single task or sits idle. A 3-processor execution schedule for our example computation (Figure 2.1) is shown in Figure 2.2. At step 3 of this example, processors p_1 and p_2 each execute a task while processor p_3 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 spawning thread can operate concurrently with the spawned thread. We consider spawned threads to be children of the thread that did the spawning, and a thread may spawn as many children as it desires. In this way, threads are organized into a *spawn tree* as indicated in Figure 2.1 by the downward-pointing, shaded edges, called *spawn* edges, that connect threads to their spawned children. The spawn tree is the parallel analogue of a call tree. In our example computation, the spawn tree's *root* thread Γ_1 has two children, Γ_2 and Γ_6, and thread Γ_2 has three children, Γ_3, Γ_4, and Γ_5. Threads Γ_3, Γ_4, Γ_5, and Γ_6, which have no children, are *leaf* threads.

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. An execution

step	living threads						processor activity p_1	p_2	p_3
1	$\mathbf{\Gamma_1}$						v_1		
2	$\mathbf{\Gamma_1}$						v_2		
3	$\mathbf{\Gamma_1}$	$\mathbf{\Gamma_2}$					v_3	v_{14}	
4	Γ_1	$\mathbf{\Gamma_2}$	$\mathbf{\Gamma_3}$			$\mathbf{\Gamma_6}$	v_4	v_6	v_{15}
5	Γ_1	$\mathbf{\Gamma_2}$	$\mathbf{\Gamma_3}$	$\mathbf{\Gamma_4}$		$\mathbf{\Gamma_6}$	v_5	v_9	v_{16}
6	$\mathbf{\Gamma_1}$	Γ_2		$\mathbf{\Gamma_4}$	$\mathbf{\Gamma_5}$	$\mathbf{\Gamma_6}$	v_7	v_{10}	v_{17}
7	$\mathbf{\Gamma_1}$	Γ_2		$\mathbf{\Gamma_4}$	Γ_5		v_8	v_{18}	
8	$\mathbf{\Gamma_1}$	Γ_2			$\mathbf{\Gamma_5}$			v_{19}	v_{11}
9	Γ_1	Γ_2			$\mathbf{\Gamma_5}$				v_{12}
10	Γ_1	$\mathbf{\Gamma_2}$							v_{13}
11	$\mathbf{\Gamma_1}$								v_{20}

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

schedule must obey this edge in that no processor may execute a task in a spawned child thread until after the spawning task in the parent thread has been executed. In our example computation (Figure 2.1), due to the spawn edge (v_6, v_7), task v_7 cannot be executed until after the spawning task v_6. Consistent with our unit-time model of tasks, a single task may spawn at most one child. When the spawning task executes, it allocates an activation frame for the new child thread. Once a thread has been spawned and its frame has been allocated, we say the thread is *alive* or *living*. When the last task of a thread executes, it deallocates its frame and the thread *dies*. In our 3-processor execution schedule (Figure 2.2), thread Γ_5 is spawned at step 5 and dies at step 9. Therefore, it is living at steps 6, 7, 8, and 9.

An execution schedule must respect one more kind of dependency. 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. 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*, which *enables* the consuming thread to resume with its execution; the thread becomes *ready*. For example, at step 4 of our 3-processor execution schedule (Figure 2.2), thread Γ_1 is stalled at task v_{18} because task v_9 has not yet been executed. At step 5 task v_9 is executed by processor p_2, thereby enabling thread Γ_1. At step 6, thread Γ_1 is ready at task v_{18}. A multithreaded computation does not model the mechanism by which data dependencies get resolved or unresolved dependencies get detected.

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

We make the simplifying assumption that a parent thread remains alive until all its children die, 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 living threads at that time, and the total space used in executing a computation is the maximum such value over the course of the execution.

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

The notion of an execution schedule is independent of any real machine characteristics. An execution schedule simply requires that no processor executes more than one task per time step and every task is executed at a time step after all of its predecessor tasks (which connect to it via continue, spawn, or data-dependency edges) have been executed. A given execution schedule may not be viable for a real machine, since the schedule may not account for properties such as communication latency. For example, in our 3-processor execution schedule (Figure 2.2), task v_{11} is executed at step 8 by processor p_3 exactly one step after v_8 is executed by processor p_1, even though there is a data dependency between them that surely requires some latency to be resolved.

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

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 dag of tasks. 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.

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 $T(\mathcal{X})$ denote the time to execute the computation using P-processor execution schedule $\mathcal{X}$, and let

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

denote the minimum execution time with P processors—the minimum being taken over all P-processor 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, we have $T_P \geq T_1/P$. Of course, we also have $T_P \geq T_\infty$. Early work by Graham [20, 21] and independently by Brent [11, Lemma 2] yields the bound $T_P \leq T_1/P + T_\infty$. The following theorem extends these results minimally to show that this upper bound on T_P can be obtained by *greedy schedules*, i.e., 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.

THEOREM 3.1 (the greedy-scheduling theorem). *For any multithreaded computation with work T_1 and computation depth T_∞, and for any number P of processors, every greedy P-processor execution schedule $\mathcal{X}$ achieves $T(\mathcal{X}) \leq T_1/P + T_\infty$.*

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 P-processor execution schedule $\mathcal{X}$ where the set of tasks executed at time i, for $i = 1, 2, \ldots, k$, is denoted $\mathcal{V}_i$, with $k = T(\mathcal{X})$. The $\mathcal{V}_i$ 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 $V_i = V_{i-1} - \mathcal{V}_i$, 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 $|\mathcal{V}_i|$. Consider a step i with $|\mathcal{V}_i| = P$. 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 $|\mathcal{V}_i| < P$. In this case, since $\mathcal{X}$ is greedy, $\mathcal{V}_i$ 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 $|\mathcal{V}_i| < P$.

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

The greedy-scheduling theorem (Theorem 3.1) 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, the greedy-scheduling theorem tells us when we can obtain *linear parallel speedup*, that is, when we can find an execution schedule $\mathcal{X}$ such that $T(\mathcal{X}) = \Theta(T_1/P)$. 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 $T(\mathcal{X}) \leq 2T_1/P$. 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 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 the greedy-scheduling theorem and consider whether there exist execution schedules that achieve similar time bounds while also making efficient use of space. Of course, we will have to quantify a space bound to capture what we mean by "efficient use of space."

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 is just a spawn tree of threads. 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.

We shall denote the space required by a P-processor execution schedule $\mathcal{X}$ of a multithreaded computation by $S(\mathcal{X})$. Recall that $S(\mathcal{X})$ is just the maximum, over all steps in $\mathcal{X}$, of the sum of the sizes of the activation frames of the living 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 $S_1 \leq S(\mathcal{X})$, where $S_1 = \min_{\mathcal{X}} S(\mathcal{X})$ denotes the minimum space used by a 1-processor execution.

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

THEOREM 3.2. *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 we have $S(\mathcal{X}) \geq \mathcal{A}$, and more specifically, we have $S_1 \geq \mathcal{A}$.*

Proof. In any schedule, the leaf thread with greatest activation depth must be alive at some time step. Since we assume that if a thread is alive, its parent is alive, when the deepest leaf thread is alive, all its ancestors are alive, and hence, all its ancestors' frames are allocated. However, the sum of the sizes of its ancestors' activation frames is just the activation depth. Since $S(\mathcal{X}) \geq \mathcal{A}$ holds for all P-processor schedules $\mathcal{X}$ 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 living 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 spawn tree always visits all the tasks on which a given task depends before it visits the given task. In the example computation of Figure 2.1, the left-to-right depth-first search order is $v_1, v_2, \ldots, v_{20}$, and this computation is depth-first. 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 dies. If the thread spawns a child, the parent thread is put aside to be resumed only after the child thread dies; the scheduler then begins work on the child, executing the

child until it either spawns a child or dies.

THEOREM 3.3. *For any depth-first computation, we have* $S_1 = \mathcal{A}$.

Proof. At any time in a serial depth-first execution of the computation, the set of living threads always forms a path from the root. Therefore, the space required is just the activation depth of the computation. By Theorem 3.2, $S_1 \geq \mathcal{A}$, and thus the the space used is the minimum possible. □

The remainder of this paper considers only depth-first computations, and we shall use S_1 to denote a computation's activation depth.

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

Even for depth-first computations, a P-processor schedule may use nearly P times the space of a 1-processor schedule. Consider, for example, a computation in which the root thread is a loop that spawns a child thread for each iteration. A single processor executing this computation uses only the space needed for a single iteration (plus the space used by the root), since upon completion of an iteration, all the memory can be freed and then reused for the next iteration. A natural P-processor execution, however, might execute P iterations concurrently, thereby requiring the memory of P iterations. Such a P-processor execution schedule $\mathcal{X}$ uses space $S(\mathcal{X}) = \Theta(S_1 P)$.

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 value independent of P:

$$\begin{aligned} T(\mathcal{X})S(\mathcal{X}) &= O(T_1/P) \cdot O(S_1 P) \\ &= O(T_1 S_1) \ . \end{aligned}$$

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

4. Lower bound. In this section 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 S_1 and any (reasonably large) serial execution time T_1, we can exhibit a depth-first multithreaded computation with work T_1 and activation depth S_1 but with provably bad time/space tradeoff characteristics. Being depth-first, we know from Theorem 3.3 that our computation can be executed using serial space S_1. Furthermore, we know from the greedy-scheduling theorem (Theorem 3.1) that for any number P of processors, any greedy P-processor execution schedule $\mathcal{X}$ achieves $T(\mathcal{X}) \leq T_1/P + T_\infty$. Our computation has computation depth $T_\infty \approx \sqrt{T_1}$, and consequently, for $P = O(\sqrt{T_1})$, a greedy P-processor schedule $\mathcal{X}$ yields $T(\mathcal{X}) = O(T_1/P)$, i.e., linear speedup. We show, however, that any P-processor schedule $\mathcal{X}$ achieving $T(\mathcal{X}) = O(T_1/P)$ must use space $S(\mathcal{X}) = \Omega(\sqrt{T_1}(P-1))$. Of course, $\sqrt{T_1}$ may be much larger than S_1, and hence, this space bound is nowhere near linear in its space expansion.

THEOREM 4.1. *For any amount of serial space $S_1 \geq 4$ and serial time $T_1 \geq 16S_1^2$, there exists a depth-first multithreaded computation with work T_1, computation depth $T_\infty \leq 8\sqrt{T_1}$, and activation depth S_1 with the following property: for any number P of processors and any value ρ in the range $1 \leq \rho \leq (1/8)T_1/T_\infty$, if $\mathcal{X}$ is a P-processor execution schedule that achieves speedup ρ—that is, $T(\mathcal{X}) \leq T_1/\rho$— then the schedule requires space $S(\mathcal{X}) \geq (1/4)(\rho - 1)\sqrt{T_1} + S_1$.*

Proof. To exhibit a depth-first multithreaded computation with work T_1, computation depth T_∞, and activation depth S_1, we first ignore the partitioning of tasks into threads and consider just the dag structure of the computation. Minus a few tasks and dependencies, the dag appears as in Figure 4.1(a). The tasks are organized into

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

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

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

tasks that we call *headers* (vertical hashed in Figure 4.1(a)). After the headers, each chain contains

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

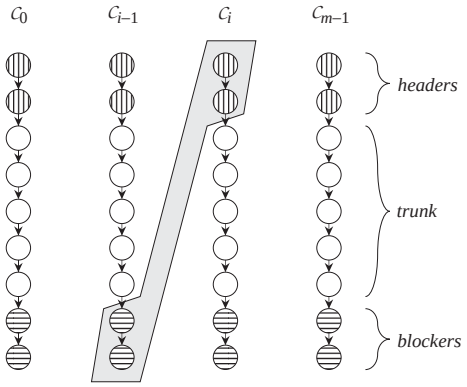
tasks (plain white in Figure 4.1(a)) that form the *trunk*. At the end of each chain, there are λ *blockers* (horizontal hashed in Figure 4.1(a)). Each chain, therefore, consists of $2\lambda + \nu = 2(\sqrt{T_1}/S_1) + 6\sqrt{T_1}$ tasks. Since there are $m = \sqrt{T_1}/8$ chains, the total number of tasks accounted for by the m chains is $(2\sqrt{T_1}/S_1 + 6\sqrt{T_1})\sqrt{T_1}/8 = (3/4)T_1 + (1/4)T_1/S_1$, and this number is no more than $(13/16)T_1$ since $S_1 \geq 4$. The remaining (at least) $(3/16)T_1$ tasks form the parts of the computation not shown in Figure 4.1(a).

There are no dependencies between different chains, so the average available parallelism T_1/T_∞ is at least $m = \sqrt{T_1}/8$ and the computation depth T_∞ is no more than $8\sqrt{T_1}$ as promised.

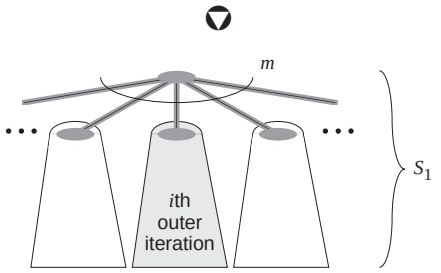
Now, consider the partitioning of the tasks from each chain into the actual threads. As alluded to in Figure 4.1(b), the root thread has m child threads, each of which is the root of a subcomputation that we call an *outer iteration*. (The outer iterations contain *inner iterations* that will be discussed later.) Each of these outer iterations contains $\sqrt{T_1}/2$ threads. As illustrated by the shading in Figure 4.1, the ith outer iteration for $i = 1, 2, \ldots, m-1$ contains both the header tasks of chain $\mathcal{C}_i$ and the blocker tasks of chain $\mathcal{C}_{i-1}$. These tasks are organized into the threads of the outer iteration so as to ensure that chain $\mathcal{C}_i$ cannot begin executing its trunk tasks until all $\sqrt{T_1}/2$ of the outer iteration's threads have been spawned, and none of these threads can die until chain $\mathcal{C}_{i-1}$ begins executing its blocker tasks. (We will exhibit this organization later.) Thus, if chain $\mathcal{C}_i$ begins executing its trunk tasks before chain $\mathcal{C}_{i-1}$ finishes its, then the execution will require at least $\sqrt{T_1}/2$ space.

For any number P of processors, consider any valid P-processor execution schedule $\mathcal{X}$. For each chain $\mathcal{C}_i$, let $t_i^{(s)}$ denote the time step at which $\mathcal{X}$ executes the first

[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.



(A) *Chains of tasks.*



(B) *Outer iterations.*

FIG. 4.1. *Constructing a computation with no efficient execution schedule. The header tasks of chain $\mathcal{C}_i$ and the blocker tasks of chain $\mathcal{C}_{i-1}$ are both placed in the threads of the ith outer iteration.*

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 chains, $\mathcal{C}_i$ 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 $\sqrt{T_1}/2$ space. In this case, we consider the time interval from $t_i^{(\mathrm{s})}$ (inclusive) to $t_{i-1}^{(\mathrm{f})}$ (exclusive) during which we say that chain $\mathcal{C}_i$ is *exposed*, and we let $\tau_i = t_{i-1}^{(\mathrm{f})} - t_i^{(\mathrm{s})}$ denote the amount of time chain $\mathcal{C}_i$ is exposed. See Figure 4.2. If $t_i^{(\mathrm{s})} \geq t_{i-1}^{(\mathrm{f})}$ then chain $\mathcal{C}_i$ is never exposed and we let $\tau_i = 0$. As we have seen, over the time interval during which a chain is exposed, it uses at least $\sqrt{T_1}/2$ space. We will show that in order to achieve speedup ρ—that is $T(\mathcal{X}) \leq T_1/\rho$—there must be some time step during the execution at which at least $\lceil (3/4)\rho \rceil - 1$ chains are exposed.

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

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

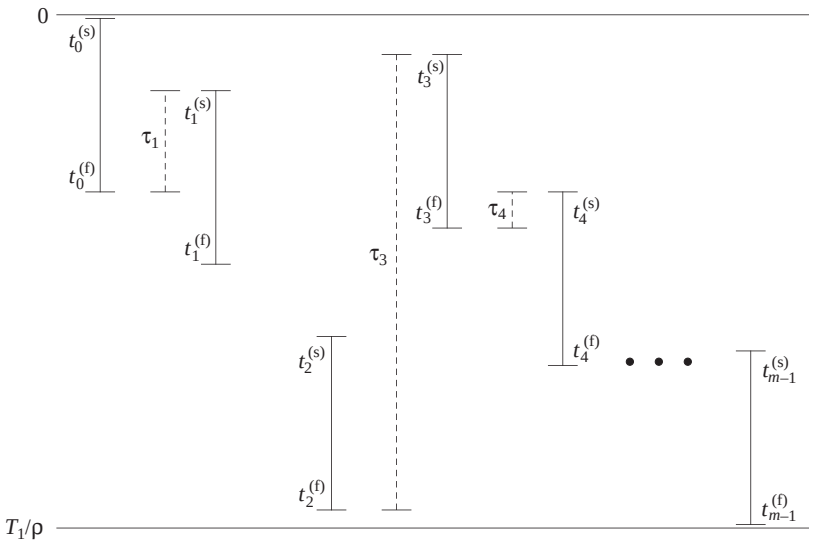


FIG. 4.2. *Scheduling the execution of the chains. A solid vertical interval from $t_i^{(s)}$ to $t_i^{(f)}$ indicates the time during which the trunk of chain $\mathcal{C}_i$ is being executed. When $t_i^{(s)} < t_{i-1}^{(f)}$, we can define an interval, shown dashed, of length $\tau_i = t_{i-1}^{(f)}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

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

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

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,

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

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

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

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

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

Then the average number of exposed chains per time step is

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

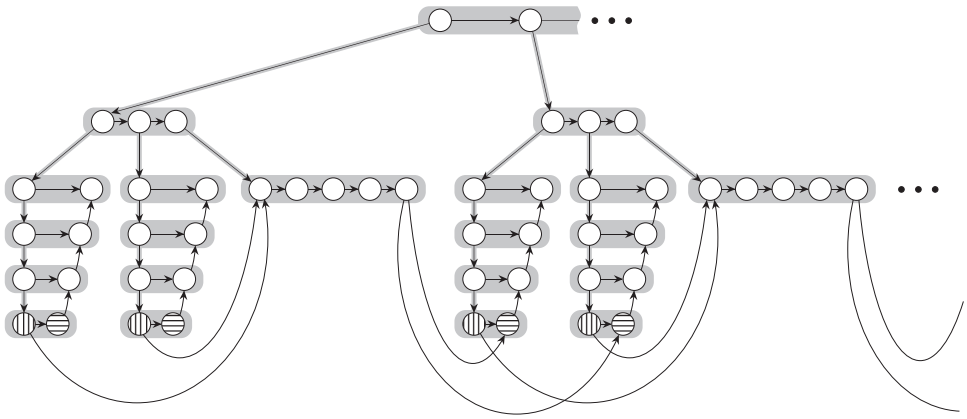


FIG. 4.3. *Laying out the chains into the threads of a multithreaded computation. As before, the header tasks are vertical hashed, and the blocker tasks are horizontal hashed. In this example, each activation frame has unit size so* $S_1 = 6$. *Also, in this example* $\lambda = 2$, $\nu = 5$, *and only the first* 2 *out of the* m *tasks in the root thread are shown. Each task of the root thread spawns a child (an outer iteration), and each child thread contains* $\lambda + 1 = 3$ *tasks; the first* λ *of these spawn a child thread which is the root of an inner iteration with activation depth* $S_1 - 2 = 4$, *and the last one spawns a leaf thread with the* $\nu = 5$ *trunk tasks of a single chain.*

$$\begin{aligned} &\geq \frac{1}{T_1/\rho}\left(m\nu - T_1/\rho\right) \\ &= \frac{3}{4}\rho - 1, \end{aligned}$$

since $m = \sqrt{T_1}/8$ and $\nu = 6\sqrt{T_1}$. There must be some time step t^* for which $\mathit{exposed}(t^*)$ 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 chain uses space $\sqrt{T_1}/2$, we have

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

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

All that remains is exhibiting the organization of the tasks of each chain into a depth-first multithreaded computation with work T_1, computation depth $T_\infty \leq 8\sqrt{T_1}$, and activation depth S_1 in such a way that each exposed chain uses $\sqrt{T_1}/2$ space. There are actually many ways of creating such a computation. One such way, which 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 (an outer iteration). Each child thread contains $\lambda + 1$ tasks; the first λ of these spawn a child thread which is the root of a subcomputation that we call an *inner iteration*. Each inner iteration has 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 chain. Each of these inner iterations contains a single header from one chain and a single blocker from the previous chain (except in the case of the first group of λ) as shown in Figure 4.3. The header and blocker in an inner iteration are organized such that in order to execute the header, all $S_1 - 2$ of the threads in the inner iteration must be spawned, and none of them can die until the blocker executes. Thus, when a chain is exposed, all λ of these inner iterations have all of their threads living, thereby using space $\lambda(S_1 - 2) \geq (\sqrt{T_1}/S_1)(S_1/2) = \sqrt{T_1}/2$.

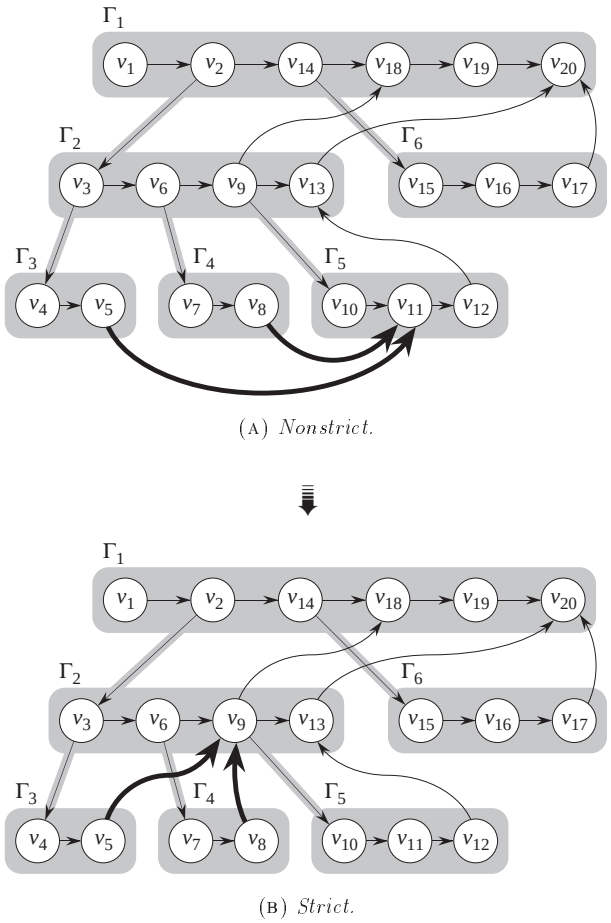
We can verify from Figure 4.3 and from the given values of m, λ, and ν that this construction actually has work slightly less than T_1; in order to make the work equal to T_1 we can just add the extra tasks evenly among the threads that contain the trunk of each chain (thereby increasing ν by a bit). 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 restrictions 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 4.1 holds even for multithreaded computations with any or all of these restrictions.

5. Scheduling algorithms for strict multithreaded computations. In the view of negative results from section 4, we consider scheduling algorithms for a specific class of depth-first multithreaded computations called "strict" computations. In this section, we show that for any strict multithreaded computation and any number P of processors, there exists a P-processor execution schedule $\mathcal{X}$ that achieves time $T(\mathcal{X}) \leq T_1/P + T_\infty$. We give two algorithms to compute such a schedule. We conclude this section by showing how some nonstrictness can be allowed in an otherwise strict computation in a way that may improve performance, but which does not adversely affect our asymptotic time and space bounds.

Given a multithreaded computation, a scheduling algorithm for a P-processor parallel computer must compute a 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 living 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.

To cope with the lower bound from Theorem 4.1, we now restrict our attention to those multithreaded computations in which every data dependency goes from a thread to one of its ancestors in the spawn tree. It turns out that requiring all data dependencies to go from a thread to one of its ancestors can be viewed as requiring that all function invocations (in a functional language) be strict, and therefore, we refer to this class of computations as *strict* multithreaded computations. For example, the computation shown in Figure 5.1(a) is not strict since the bold data dependencies



(A) *Nonstrict.*

(B) *Strict.*

FIG. 5.1. (A) *This multithreaded computation (the same as Figure* 2.1*) is nonstrict since it has nonstrict data dependencies (shown bold) that go to nonancestor threads.* (B) *If we replace the nonstrict data dependencies with new strict ones (shown bold) we obtain a strict computation since all data dependencies go from a child thread to an ancestor thread.*

violate the strictness condition just stated, but by promoting these dependencies we obtain the strict computation shown in Figure 5.1(b).

Strict multithreaded computations are depth-first computations, since no data dependency can go between two distinct subcomputations of a thread. Once a thread Γ has been spawned in a strict computation, 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 dies, 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.

Algorithm GDF (which stands for *global depth-first*) maintains all living threads in a global queue prioritized by activation depth, i.e., 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

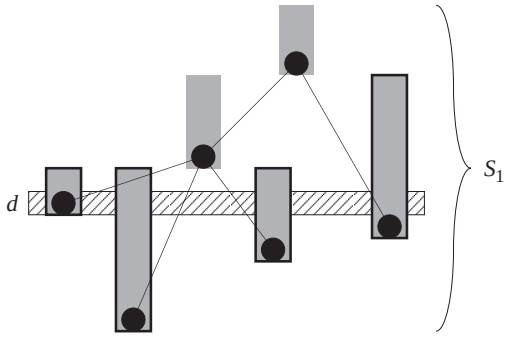


FIG. 5.2. *The spawn tree corresponding to the example computation of Figure* 2.1. *The bold outlined threads span depth d.*

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 5.1. *For any number P of processors and any strict multithreaded computation with work T_1, computation depth T_∞, and activation depth S_1, Algorithm* GDF *computes a P-processor schedule $\mathcal{X}$ that uses space $S(\mathcal{X}) \leq S_1 P$ and time $T(\mathcal{X}) \leq T_1/P + T_\infty$.*

Proof. The time bound follows immediately from the greedy-scheduling theorem (Theorem 3.1), 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 otherwise) 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 $\mathcal{A}(\Gamma') < d$. For example, Figure 5.2 depicts the spawn tree corresponding to the computation of Figure 2.1. Each thread has height equal to the size of its activation frame and is located so that the top of its activation frame is aligned with the bottom of its parent's activation frame. In this way, each black node is located at its thread's activation depth, and the bold outlined threads span depth d. For any time step t during the execution and any activation depth d, let $s(t, d)$ denote the number of living 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 shall 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 living 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 $s(t, d)$ living 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 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 $s(t,d) + (P - s(t,d)) = P$ living threads that span activation depth d. Therefore, $s(t+1,d) \leq P$, and the induction is complete. □

Algorithm GDF′ is a refinement of Algorithm GDF that achieves greater efficiency by reducing the number of accesses to the global queue. Algorithm GDF′ begins with the root thread assigned to some arbitrary processor and the global queue empty. On subsequent steps, GDF′ has completed a "previous" step and must schedule threads for a "current" step. Suppose the previous step ends with P' out of the P processors not having a thread. To start the current step, the scheduler removes from the queue the P' deepest ready threads, or, if there are fewer than P' ready threads, it removes them all. It assigns these threads arbitrarily to the P' idle processors so that each idle processor receives at most one thread. The current step is now ready to proceed. Each of the P processors that has an assigned thread executes one task from that thread. Unless that thread spawns, dies, or stalls, the processor will have a thread at the end of the current step. If the thread stalls, then the processor must return it to the global queue, and consequently, the processor will not have a thread at the end of the current step. Similarly, if the thread dies, then the processor will not have a thread at the end of the step. Lastly, if the thread spawns a child, then the processor returns the parent thread (the one it was working on) to the global queue and keeps the child thread; in this case, the processor will still have a thread at the end of the current step.

Algorithm GDF′ achieves the same performance bounds as proved in Theorem 5.1, but it requires access to the global queue only when threads spawn, die, or stall.

THEOREM 5.2. *For any number P of processors and any strict multithreaded computation with work T_1, computation depth T_∞, and activation depth S_1, Algorithm* GDF′ *computes a P-processor schedule $\mathcal{X}$ that uses space $S(\mathcal{X}) \leq S_1 P$ and time $T(\mathcal{X}) \leq T_1/P + T_\infty$.*

Proof. This proof follows the proof of Theorem 5.1, but we add the following assertion to the induction hypothesis: for any activation depth d, if a step t begins with $s(t,d) \leq P$ living 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. □

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. The next section presents and analyzes such a distributed algorithm.

We have been able to relate resource requirements to nonstrictness in the computation by characterizing two extremes. At one end, we have shown that arbitrary uses of nonstrictness make efficient execution impossible. At the other end, purely strict computations allow near optimally efficient executions. We now mention two minor results that begin to characterize resource requirements for limited uses of nonstrictness.

Given an arbitrary depth-first computation, any of the scheduling algorithms for strict computations can be employed by first adding data-dependency edges to make the computation strict. This transformation, known as *strictifying* (see Figure 5.1),

is always valid for depth-first computations. Of course, strictifying may dramatically reduce the average available parallelism, and therefore, we would like some way of exploiting the parallelism available through nonstrict spawns. 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 only 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. Unfortunately, the naive application of this rule can actually result in an execution that takes longer than the purely strict execution.

With due care, however, we can modify this rule to allow some nonstrict spawns while still guaranteeing the time and space bounds of a purely strict execution. For example, we can restrict the application of this rule to a set of threads designated by the programmer. If the programmer can designate this set of threads so as to ensure that, during execution, at most x nonstrictly spawned threads simultaneously span a given depth, then Algorithm GDF can achieve space bounded by $S_1(P + x)$ and linear speedup as in Theorem 5.1; similar results apply for Algorithm GDF$'$ and for the distributed algorithm that will be presented in the next section. Alternatively, by "sequestering" the nonstrictly spawned threads, the scheduler itself can budget the nonstrict spawns and achieve these same time and space bounds; details can be found in [4].

6. Distributed scheduling algorithms. In a distributed thread-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, dies, 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 nonlocal queues in order to facilitate some type of load balancing.

The technique of Karp and Zhang [28] suggests a randomized algorithm in which threads are located in random queues in order to achieve some balance. We can show, however, that the naive adoption of this technique does not work. In particular, threads must migrate occasionally and some degree of synchronization is needed to avoid the large deviations that result if this random process is run over a long period of time. Further discourse on these problems can be found in [4]. 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 (which stands for *local depth-first*) 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 which 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

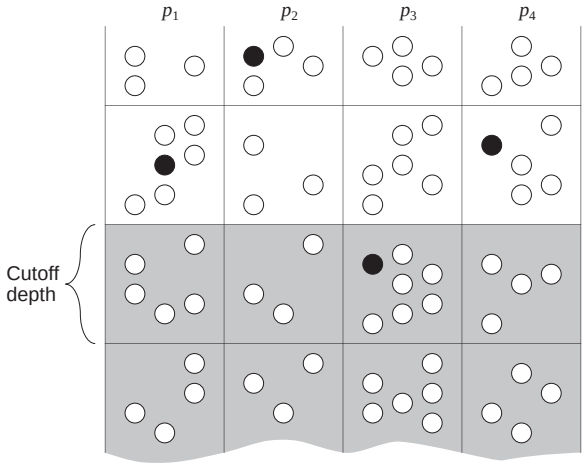


FIG. 6.1. *Computing the cutoff depth. Each column represents the local priority queue of a processor, and each row represents an activation depth with depth increasing in the downward direction. We depict each ready thread by a circle located in its processor's queue and at its activation depth. (Within a processor's queue, the horizontal ordering of threads is irrelevant.) The ready threads in each queue are ordered by activation depth with ties broken arbitrarily—this tie breaking is depicted by the vertical ordering of threads within an activation depth. In this example, the synchronization parameter* $r = 12$*, and the* 12*th deepest ready thread for each processor is shown in black (just count up from the bottom). 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.*

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

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

as illustrated in Figure 6.1.

During the computation phase of LDF(r), each processor executes at least 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 6.1. *For any number* P *of processors and any strict multithreaded computation with activation depth* S_1*, Algorithm* LDF(r) *computes a* P*-processor schedule* $\mathcal{X}$ *such that* $S(\mathcal{X}) \le 2rS_1P$.

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

notation used in the proof of Theorem 5.1, 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 straightforward.

For any activation depth d and any iteration t of the execution, we consider two cases. In the first case, suppose iteration t begins with $rP \leq s(t,d) \leq 2rP$ living 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 pigeonholing, 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 living threads spanning depth d than it started with. Now, suppose iteration t begins with $s(t,d) < rP$ living threads spanning depth d. In this case, during the computation phase, since each processor is only allowed r spawns, the number of living threads that span depth d can increase by at most rP, and therefore, the iteration ends with no more than $2rP$ living 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 6.1 is directly proportional to r. By setting $r = 6 \lg P$, the space bound of Lemma 6.1 becomes $S(\mathcal{X}) \leq 12 S_1 P \lg P$, 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 $r = 6 \lg P$, it is highly likely that each iteration succeeds.

LEMMA 6.2. *For any number P of processors and any iteration of Algorithm* LDF$(6 \lg P)$*, the iteration fails with probability no more than P^{-5}.*

Proof. 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 are deeper than the cutoff depth. Therefore, an iteration 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, and 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, yielding

$$\begin{aligned}\Pr\{Z > 6\lg P\} &\leq P \cdot \Pr\{Z_i > 6\lg P\} \\ &\leq P^{-5}\,. \qquad \square\end{aligned}$$

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 synchrony 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 6.3. *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 P-processor 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 $b = (1/3)\log_P(1/\epsilon)$. 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 we obtain

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

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 $b = (1/3)\log_P(1/\epsilon)$. Thus, with probability at least $1 - \epsilon$, we have $f = O(T_1/(P \lg P) + \log_P(1/\epsilon))$.

Now consider the successful iterations. 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 the greedy-scheduling theorem (Theorem 3.1), 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 6.1 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 6.4. *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

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

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 as follows:

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

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 $j = e^{k+1} \ln P - 1$ as follows:

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

Now we can bound $\Pr\left\{X_i \ge e^k \ln P\right\}$ by the same technique as in Lemma 6.2, since X_i has the same distribution as the random variable Z considered in the proof of Lemma 6.2:

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

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 inequalities (6.5) and (6.7) back into equation (6.4), producing

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

Finally, by 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

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

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

Proof. The space bound follows directly from Lemma 6.1 with synchronization parameter $r = 6 \lg P$. The time $T(\mathcal{X})$ is the total time taken in computation phases. Let the random variable Y denote the number of iterations. Then we can decompose $T(\mathcal{X})$ as a sum of Y mutually independent random variables, $T(\mathcal{X}) = X_1 + X_2 + \cdots + X_Y$, 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 6.3, 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 6.4, we obtain $T(\mathcal{X}) = O(Y \lg P + \lg(1/\epsilon))$ 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 $T(\mathcal{X}) = O(T_1/P + T_\infty \lg P + \lg(1/\epsilon))$. $\square$

COROLLARY 6.6. *For any number $P \ge 2$ of processors and any strict multithreaded computation with work T_1 and computation depth T_∞, Algorithm* LDF$(6 \lg P)$

computes a P-processor schedule $\mathcal{X}$ with expected execution time $\mathrm{E}[T(\mathcal{X})] = O(T_1/P + T_\infty \lg P)$.

Proof. Just use $\epsilon = 1/P$ in Theorem 6.5 to get $T(\mathcal{X}) = O(T_1/P + T_\infty \lg P)$ with probability at least $1 - 1/P$. Then we have

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

This 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 [42]. The space bound $S(\mathcal{X}) = O(S_1 P \lg P)$ indicates that Algorithm LDF$(6 \lg P)$ requires memory to scale sufficiently to allow each *physical* processor enough space to simulate $\Theta(\lg P)$ *virtual* processors. Given this much space, the time bound $\mathrm{E}[T(\mathcal{X})] = O(T_1/P + T_\infty \lg P)$ then demonstrates linear expected speedup provided the computation has $\lg P$ slack in the average available parallelism.

The space bound of Theorem 6.5 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 living 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 living thread. Since 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 $O(rS_1)$.

If we consider any given processor p and any given iteration t of the algorithm, then we can let W denote the total space being used by activation frames located in the memory of processor p. We can decompose W as a weighted sum of independent indicator random variables and show that $\mathrm{E}[W] \le 2rS_1$. Then, using a theorem due to Raghavan [36, Theorem 1], we can show that with probability at least $1 - e^{-2r}$, we have $W \le 2erS_1$.

With this probabilistic bound on the space used by a given processor at a given iteration, we can show that with appropriate choice of the synchronization parameter r, we can bound the per-processor memory by simply rerandomizing thread locations any time a processor's memory fills up. In particular, if we choose $r = \Theta(\lg P + \lg S_1)$, then the total time spent rerandomizing is $O(T_1/P)$ and the per-processor storage bound is $O(S_1(\lg P + \lg S_1))$. Details can be found in [4].

7. Related and future work. Although the work we have presented here provides some theoretical underpinnings for understanding the resource requirements of multithreaded computations, much remains to be done. In this section, we review some of the related work, both theoretical and empirical, on scheduling dynamic computations. We discuss the class of "thread-stealing" algorithms and present some of our preliminary research on this kind of scheduling algorithm.

Substantial research has been reported in the theoretical literature concerning dynamic computations. In contrast to our research on multithreaded computations, however, other theoretical research has tended to treat the aggregate resource requirements of a computation as a given, rather than as a quantity that depends on the

execution schedule. Thus, the relevant issue in this work is how to balance the load across processors. Important work in this area includes a randomized work-stealing algorithm for load balancing [38]; dynamic tree-embedding algorithms [3, 31]; and algorithms for backtrack search [27, 37, 45], which can be viewed as a multithreaded computation with no data-dependency edges. Although this work ignores aggregate space requirements, it is interesting to note that Zhang's work-stealing algorithm for backtrack search [45] actually gives at most linear expansion of space, but he does not mention this fact.

The problem of storage management for multithreaded computations has been a growing concern among practitioners [13, 22]. 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 for multithreaded computations for which simultaneously good time and space bounds have been proved.

The storage management problem can often 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. In order to curb the excessive use of space exhibited by fair scheduling, researchers from the dataflow community have developed heuristics to explicitly manage storage [15, 39]. The effectiveness of these heuristics is documented with encouraging empirical evidence but no provable time bounds.

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 serially executed programs.

Other researchers have also addressed the storage issue by attempting to relate parallel storage requirements to serial storage requirements. Burton and Sleep [12] and Halstead [22], for example, considered unfair scheduling policies based on *thread stealing*. In these thread-stealing strategies, each processor works depth-first—just like a serial execution—but when a processor runs out of ready threads, it *steals* threads from other processors. In many cases, this scheduling policy results in each processor using no more space than that used by a single processor, but a problem arises as to what to do when all threads in a processor have stalled. If the processor goes out to steal a thread from another processor, greater-than-linear space expansion may result. If the processor goes idle, however, linear speedup is not guaranteed. For these unfair scheduling policies, characterizing the performance analytically is difficult.

Thread stealing has also been employed in two parallel chess-playing programs. Zugzwang [18] is a program in which processors steal subcomputations of a chess tree using a parallel alpha-beta search algorithm. StarTech [30] is another parallel program organized along similar lines but with a parallel scout-search algorithm. Although the authors make no guarantees of performance for their algorithms, the empirical results of these programs are good; both have won prizes in international chess competitions.

In recent work, we have obtained some preliminary results on thread stealing. We have devised a new global algorithm that forms the basis of a randomized, distributed, thread-stealing algorithm. Our new global algorithm is like GDF$'$ except for two changes. First, the global queue is not organized by activation depth; when a processor removes a ready thread from the queue, any ready thread suffices. Second, when a thread dies, the thread's processor must locate the parent thread in the global queue

and check to see if the parent has any surviving children. If the parent no longer has any surviving children, then the processor must commence work on the parent thread. Otherwise, the processor is free to take any ready thread from the global queue. It can be proved by simple induction that this algorithm satisfies the same time and space bounds as Algorithms GDF and GDF′ . In our distributed thread-stealing algorithm, we replace the global queue with local queues, one per processor. By making some generous modeling assumptions, we have been able to analyze this algorithm and to obtain bounds similar to those for Algorithm LDF. We are currently working on improving these results.

Appendix. During the time between our results becoming publicly known [7] and this journal publication, we have explored multithreaded computing more fully. We have been able to characterize the performance of a distributed thread-stealing algorithm [5, 8]. For the class of "fully strict" (well-structured) computations, this randomized algorithm achieves execution space bounded by S_1P and expected execution time bounded by $O(T_1/P+T_\infty)$, including scheduling overheads. Additionally, in contrast to Algorithm LDF, this thread-stealing algorithm is efficient with respect to communication. We have implemented this thread-stealing algorithm in the runtime system for *Cilk* [5, 6], a parallel multithreaded extension of the C language. By employing a provably efficient scheduler, Cilk is able to deliver efficient and predictable performance, guaranteed. Moreover, structure in the Cilk programming model facilitates the implementation of "adaptive parallelism" and transparent fault tolerance in a runtime system for Cilk on networks of workstations [5, 9]. More information about Cilk is available online at `http://theory.lcs.mit.edu/~cilk`.

Acknowledgments. The authors thank Bonnie Berger, Tom Cormen, Esther Jesurum, Mike Klugerman, Bradley Kuszmaul, Tom Leighton, Arthur Lent, Greg Papadopoulos, Atul Shrivastava, and Ethan Wolf of the MIT Laboratory for Computer Science for insightful discussions.

REFERENCES

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, *APRIL: A processor architecture for multiprocessing*, in Proc. 17th Annual Intl. Symposium on Computer Architecture, Seattle, WA, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 104–114; Tech. Report MIT/LCS/TM-450, MIT Laboratory for Computer Science, Cambridge, MA, 1991.

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

[3] S. Bhatt, D. Greenberg, T. Leighton, and P. Liu, *Tight bounds for on-line tree embeddings*, in Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, SIAM, Philadelphia, 1991, pp. 344–350.

[4] R. D. Blumofe, *Managing Storage for Multithreaded Computations*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992; Tech. Report MIT/LCS/TR-552, MIT Laboratory for Computer Science, Cambridge, MA, 1992.

[5] R. D. Blumofe, *Executing Multithreaded Programs Efficiently*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1995.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, in Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Santa Barbara, CA, ACM, New York, 1995, pp. 207–216.

[7] R. D. Blumofe and C. E. Leiserson, *Space-efficient scheduling of multithreaded computations*, in Proc. of the 25th Annual ACM Symposium on Theory of Computing (STOC), San Diego, CA, ACM, New York, 1993, pp. 362–371.

[8] R. D. Blumofe and C. E. Leiserson, *Scheduling multithreaded computations by work stealing*, in Proc. of the 35th Annual Symposium on Foundations of Computer Science (FOCS), Santa Fe, NM, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 356–368.

[9] R. D. Blumofe and D. S. Park, *Scheduling large-scale parallel computations on networks of workstations*, in Proc. of the Third Intl. Symposium on High Performance Distributed Computing (HPDC), San Francisco, CA, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 96–105.

[10] B. Boothe and A. Ranade, *Improved multithreading techniques for hiding communication latency in multiprocessors*, in Proc. of the 19th Annual Intl. Symposium on Computer Architecture, Gold Coast, Australia, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 214–223.

[11] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[12] F. W. Burton and M. R. Sleep, *Executing functional programs on a virtual tree of processors*, in Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture, Portsmouth, NH, ACM, New York, 1981, pp. 187–194.

[13] D. E. Culler, *Resource Management for the Tagged Token Dataflow Architecture*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1980; Technical Report MIT/LCS/TR-332, MIT Laboratory for Computer Science, Cambridge, MA, 1985.

[14] D. E. Culler, *Managing Parallelism and Resources in Scientific Dataflow Programs*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1990; Tech. Report MIT/LCS/TR-446, MIT Laboratory for Computer Science, Cambridge, MA, 1990.

[15] D. E. Culler and Arvind, *Resource requirements of dataflow programs*, in Proc. of the 15th Annual Intl. Symposium on Computer Architecture, Honolulu, HI, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 141–150; Computation Structures Group Memo 280, MIT Laboratory for Computer Science, Cambridge, MA, 1987.

[16] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, *Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine*, in Proc. of the Fourth Intl. Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, ACM, New York, 1991, pp. 164–175.

[17] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, *Architecture of a message-driven processor*, in Proc. of the 14th Annual Intl. Symposium on Computer Architecture, Pittsburgh, PA, IEEE Computer Society Press, Los Alamitos, CA, 1987, pp. 189–196.

[18] R. Feldmann, P. Mysliwietz, and B. Monien, *Game tree search on a massively parallel system*, Adv. Comput. Chess, 7 (1993), pp. 203–219.

[19] V. G. Grafe and J. E. Hoch, *The Epsilon-2 hybrid dataflow architecture*, in Proc. 35th IEEE Computer Society Intl. Computer Conf. (COMPCON 90), San Francisco, CA, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 88–93.

[20] R. L. Graham, *Bounds for certain multiprocessing anomalies*, The Bell System Tech. J., 45 (1966), pp. 1563–1581.

[21] R. L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[22] R. H. Halstead, Jr., *Multilisp: A language for concurrent symbolic computation*, ACM Trans. Prog. Lang. Syst., 7 (1985), pp. 501–538.

[23] R. H. Halstead, Jr. and T. Fujita, *MASA: A multithreaded processor architecture for parallel symbolic computing*, in Proc. of the 15th Annual Intl. Symposium on Computer Architecture, Honolulu, HI, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 443–451.

[24] W. Horwat, *Concurrent Smalltalk on the Message-Driven Processor*, Tech. Report MIT/AI/TR-1321, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1991.

[25] R. A. Iannucci, *Toward a dataflow/von Neumann hybrid architecture*, in Proc. of the 15th Annual Intl. Symposium on Computer Architecture, Honolulu, HI, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 131–140; Computation Structures Group Memo 275, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[26] S. Jagannathan and J. Philbin, *A customizable substrate for concurrent languages*, in Proc. of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, CA, ACM, New York, 1992, pp. 55–67.

[27] C. Kaklamanis and G. Persiano, *Branch-and-bound and backtrack search on mesh-connected arrays of processors*, in Proc. of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, CA, ACM, New York, 1992, pp. 118–126.

[28] R. M. Karp and Y. Zhang, *A randomized parallel branch-and-bound procedure*, in Proc. of the 20th Annual ACM Symposium on Theory of Computing, Chicago, IL, ACM, New York, 1988, pp. 290–300.

[29] S. W. Keckler and W. J. Dally, *Processor coupling: Integrating compile time and runtime scheduling for parallelism*, in Proc. of the 19th Annual Intl. Symposium on Computer Architecture, Gold Coast, Australia, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 202–213.

[30] B. C. Kuszmaul, *Synchronized MIMD Computing*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994; Tech. Report MIT/LCS/TR-645, MIT Laboratory for Computer Science, 1994; also available online via ftp://theory.lcs.mit.edu/pub/bradley/phd.ps.Z.

[31] T. Leighton, M. Newman, A. G. Ranade, and E. Schwabe, *Dynamic tree embeddings in butterflies and hypercubes*, in Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, ACM, New York, 1989, pp. 224–234.

[32] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., *Lazy task creation: A technique for increasing the granularity of parallel programs*, IEEE Trans. Parallel Distrib. Systems, 2 (1991), pp. 264–280.

[33] R. S. Nikhil and Arvind, *Can dataflow subsume von Neumann computing?*, in Proc. of the 16th Annual Intl. Symposium on Computer Architecture, Jerusalem, Israel, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 262–272; Computation Structures Group Memo 292, MIT Laboratory for Computer Science, Cambridge, MA, 1989.

[34] R. S. Nikhil, G. M. Papadopoulos, and Arvind, *$^*T$: A multithreaded massively parallel architecture*, in Proc. of the 19th Annual Intl. Symposium on Computer Architecture, Gold Coast, Australia, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 156–167; Computation Structures Group Memo 325–1, MIT Laboratory for Computer Science, Cambridge, MA, 1991.

[35] G. M. Papadopoulos and K. R. Traub, *Multithreading: A revisionist view of dataflow architectures*, in Proc. of the 18th Annual Intl. Symposium on Computer Architecture, Toronto, Canada, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 342–351; Computation Structures Group Memo 330, MIT Laboratory for Computer Science, Cambridge, MA, 1991.

[36] P. Raghavan, *Probabilistic construction of deterministic algorithms: Approximating packing integer programs*, J. Comput. System Sci., 37 (1988), pp. 130–143.

[37] A. Ranade, *Optimal speedup for backtrack search on a butterfly network*, in Proc. of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, Hilton Head, SC, ACM, New York, 1991, pp. 40–48.

[38] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, *A simple load balancing scheme for task allocation in parallel machines*, in Proc. of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, Hilton Head, SC, ACM, New York, 1991, pp. 237–245.

[39] C. A. Ruggiero and J. Sargeant, *Control of parallelism in the Manchester dataflow machine*, in Functional Programming Languages and Computer Architecture, Lecture Notes in Comput. Sci., 274, Springer-Verlag, Berlin, 1987, pp. 1–15.

[40] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura, *Thread-based programming for the EM-4 hybrid dataflow machine*, in Proc. of the 19th Annual Intl. Symposium on Computer Architecture, Gold Coast, Australia, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 146–155.

[41] K. R. Traub, *Sequential Implementation of Lenient Programming Languages*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1988; Tech. Report MIT/LCS/TR-417, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[42] L. G. Valiant, *A bridging model for parallel computation*, Comm. ACM, 33 (1990), pp. 103–111.

[43] M. T. Vandevoorde and E. S. Roberts, *WorkCrews: An abstraction for controlling parallelism*, Internat. J. Parallel Programming, 17 (1988), pp. 347–366.

[44] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, *Active messages: A mechanism for integrated communication and computation*, in Proc. of the 19th Annual Intl. Symposium on Computer Architecture, Gold Coast, Australia, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 256–266.

[45] Y. Zhang, *Parallel Algorithms for Combinatorial Search Problems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1989; Tech. Report UCB/CSD 89/543, University of California at Berkeley, Computer Science Division, 1989.