Space-Efficient Scheduling of Multithreaded Computations

(extended abstract)

Robert D. Blumofe
Charles E. Leiserson

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

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. We develop a formal graph-theoretic model of multithreaded computation and give three important measures of time and space: T_1 is the time required for executing the computation on 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 1 processor. We consider a computation executed on P processors to be time-efficient if the time is $O(T_1/P + T_\infty)$, that is, it achieves linear speedup when $P = O(T_1/T_\infty)$. We consider a P-processor execution of the computation to be space-efficient if it uses $O(S_1 P)$ total space, that is, the space per processor is within a constant factor of the space required for a 1-processor execution.

We show that there exist multithreaded computations such that no execution schedule can simultaneously achieve efficient time and efficient space. But by restricting ourselves to "strict" computations—those in which all arguments to a procedure must be available before the procedure can be invoked—we obtain much more positive results. Specifically, we show that for any strict multithreaded computation, there exists an execution schedule that achieves both efficient time and efficient space. We give a simple on-line algorithm to compute such schedules. Unfortunately, because the algorithm uses a global queue, the overhead of computing the schedule can be substantial. We show however, that this overhead problem can be overcome; we give a decentralized algorithm that can compute and execute a P-processor schedule on-line in expected time $O(T_1/P + T_\infty \lg P)$ and worst-case space $O(S_1 P \lg P)$, including overhead costs. We also indicate how some nonstrictness can be allowed in an otherwise strict computation in a way that may improve performance, but does not adversely affect the asymptotic time and space bounds.

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

1 Introduction

In the course of investigating schemes for general-purpose MIMD-style parallel computation, many diverse research groups have converged on multithreading as a dominant paradigm. As an example, modern dataflow systems [8, 11, 13, 22] 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, 4] or message arrivals in message-passing multicomputers [9, 24].

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

In attempting to expose parallelism, however, schedulers often end up exposing more parallelism than the computer can actually exploit, and since each active thread requires the use of a certain amount of memory, such schedulers can easily overrun the memory capacity of the machine [7, 12, 21]. To date, the space requirements of multithreaded computations have been managed with heuristics or not at all [7, 12, 21]. 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 $S_P(\mathcal{X})$ required by a P-processor execution schedule with the space S_1 used by a space-optimal 1-processor execution. We wish to use as little space as possible, and we argue that a space-efficient execution schedule exhibits linear expansion of space, that is, $S_P(\mathcal{X}) = O(S_1 \cdot P)$.

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

We show that for any strict multithreaded computation and any number P of processors, there exists an execution schedule $\mathcal{X}$ that achieves time $T_P(\mathcal{X}) \leq T_1/P + T_\infty$, where T_∞ is a lower bound on execution time even for arbitrarily large numbers of processors, and space $S_P(\mathcal{X}) \leq S_1 P$. Such a schedule exhibits linear expansion of space and linear speedup, $T_P(\mathcal{X}) = O(T_1/P)$, provided the average available parallelism, which we define as T_1/T_∞, is at least proportional to P, that is, $T_1/T_\infty = \Omega(P)$. We 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 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. Where proofs are omitted or sketched, details can be found in [3].

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 1, for example, each shaded block is a thread with circles representing tasks and the horizontal edges, called *continue* edges, representing the sequential ordering. The tasks of a thread must execute in this sequential order from the first (leftmost) task to the last (rightmost) task. In order to execute a thread, we allocate for it a chunk of memory, called an *activation frame*, that the tasks of the thread can use to store the values on which they compute.

An *execution schedule* for a multithreaded computation determines which processors of a parallel computer execute which tasks at each step. An execution schedule depends on the particular multithreaded computation and the number of processors in the parallel computer. In any given step of an execution schedule, each processor either executes a single task or sits idle.

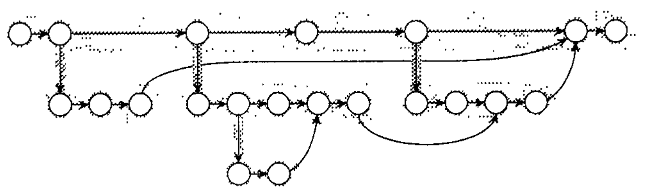

Figure 1: An example multithreaded computation.

During the course of its execution, a thread may create, or *spawn*, other threads. Spawning a thread is like a subroutine call, except that the calling routine can operate concurrently with the called routine. We consider spawned threads to be children of the thread that did the spawning. In this way, threads are organized into a tree hierarchy as indicated in Figure 1 by the shaded vertical edges, called *spawn* edges. Each spawn edge goes from a specific task, the task that actually does the spawn operation, in the parent thread to the first task of the child thread. When a thread executes its last task, it *terminates.*

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

A valid 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 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 resolves, and the consuming thread can proceed with its execution—the thread becomes *ready.*

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

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

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 1 that organizes tasks into threads, our multithreaded computation is just a directed, acyclic graph, or *dag*. We define the *work* of the computation to be the total number of tasks and the *computation depth* to be the length of a longest directed path in the dag.

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

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

denote the minimum execution time with P processors—the minimum being taken over all valid 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, $T_P \geq T_1/P$. And, of course, we also have $T_P \geq T_\infty$. Brent's Theorem [5, Lemma 2] yields the bound $T_P \leq T_1/P + T_\infty$. The following theorem extends Brent's Theorem minimally to show that this upper bound on T_P can be obtained by *greedy schedules*: those in which at each step of the execution, if at least P tasks are ready, then P tasks execute, and if fewer than P tasks are ready, then all execute.

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

Theorem 1 can be interpreted in two important ways. First, the time bound given by the theorem says that any greedy schedule yields an execution time that is within a factor of 2 of an optimal schedule, which follows because $T_1/P + T_\infty \leq 2\max\{T_1/P, T_\infty\}$ and $T_P \geq \max\{T_1/P, T_\infty\}$. Second, Theorem 1 tells us when we can obtain *linear parallel speedup*, that is, when we can find an execution schedule $\mathcal{X}$ such that $T_P(\mathcal{X}) = \Theta(T_1/P)$. Specifically, when the number P of processors is no more than the *average available parallelism* T_1/T_∞, then $T_1/P \geq T_\infty$, which implies that for a greedy schedule $\mathcal{X}$, we have $T_P(\mathcal{X}) \leq 2T_1/P$. We shall be especially interested in the regime where $P = O(T_1/T_\infty)$ and linear speedup is possible, since outside this regime, linear speedup is impossible to achieve because $T_P \geq T_\infty$.

These results on dag scheduling have been known for years. A multithreaded computation, however, adds further structure to the dag: the partitioning of tasks into threads. This additional structure allows us to quantify the space used in executing a multithreaded computation. Once we have quantified space usage, we will look back at Theorem 1 and consider whether there exist execution schedules that achieve similar time bounds while also making efficient use of space. Of course, we will have to quantify a space bound to capture what we mean by "efficient use of space."

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

We shall denote the space required by a P-processor execution schedule $\mathcal{X}$ of a multithreaded computation by $S_P(\mathcal{X})$. Recall that $S_P(\mathcal{X})$ is just the maximum, over all steps in $\mathcal{X}$, of the sum of the sizes of the activation frames of the active threads at that step. Since we can always simulate a P-processor execution with a 1-processor execution that uses no more space, we have $S_1 \leq S_P(\mathcal{X})$, where $S_1 = \min_{\mathcal{X}} S_1(\mathcal{X})$ 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 2 *Let $\mathcal{A}$ be the activation depth of a multithreaded computation, and let $\mathcal{X}$ be a P-processor execution schedule of the computation. Then $S_P(\mathcal{X}) \geq \mathcal{A}$, and more specifically, $S_1 \geq \mathcal{A}$.* ∎

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

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

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

We now turn our attention to determining how much space $S_P(\mathcal{X})$ a P-processor execution schedule $\mathcal{X}$ can use and still be considered efficient with respect to space usage. Our strategy is to compare the space used by a P-processor schedule with the space required by an optimal 1-processor schedule. Of course, we can always ignore $P - 1$ of the processors 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 $S_P(\mathcal{X}) = \Theta(S_1 P)$.

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

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

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 that our computation can be executed using serial space S_1. Furthermore, we know from Theorem 1 that for any number P of processors, any greedy P-processor execution schedule $\mathcal{X}$ achieves $T_P(\mathcal{X}) \leq T_1/P + T_\infty$. Our computation has computation depth $T_\infty \approx \sqrt{T_1}$, and consequently, for $P = O(\sqrt{T_1})$, a greedy schedule $\mathcal{X}$ yields $T_P(\mathcal{X}) = O(T_1/P)$—linear speedup. We show, however, that any schedule achieving $T_P(\mathcal{X}) = O(T_1/P)$ must use space $S_P(\mathcal{X}) = \Omega(\sqrt{T_1}(P-1))$. Of course, $\sqrt{T_1}$ may be much larger than S_1, hence, this space bound is nowhere near linear in its space expansion.

Theorem 4 *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, such that for any number P of processors and any value ρ in the range $1 \leq \rho \leq \frac{1}{8}T_1/T_\infty$, if $\mathcal{X}$ is a valid P-processor execution schedule that achieves $T_P(\mathcal{X}) \leq T_1/\rho$, then $S_P(\mathcal{X}) \geq \frac{1}{4}(\rho - 1)\sqrt{T_1} + S_1$.*

Proof sketch: 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 2(a). The tasks are organized into

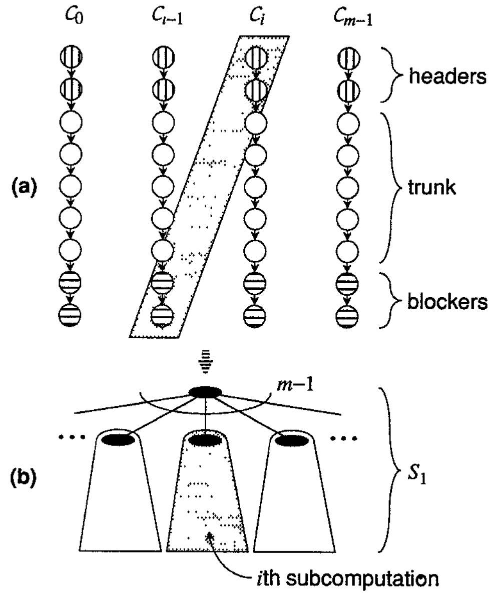

Figure 2: Constructing a nasty computation.

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

separate components $C_0, C_1, \ldots, C_{m-1}$ that we call *chains.*[1] Each chain begins with

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

tasks that we call *headers* (vertically hashed in Figure 2(a)). After the headers, each chain contains

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

tasks (plain white in Figure 2(a)) that form the *trunk.* At the end of each chain, there are λ *blockers* (horizontally hashed in Figure 2(a)).[2] 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.

To complete the description of this computation, we consider the partitioning of the tasks from each chain into the actual threads. As alluded to in Figure 2(b), the root thread has $m - 1$ child threads, each of which is the root of a subcomputation. Each of these subcomputations contains $\sqrt{T_1}/2$ threads and each of these threads has a unit-size activation frame. As illustrated by the shading in Figure 2, the ith subcomputation for $i = 1, 2, \ldots, m-1$ contains both the header tasks of chain C_i and the blocker tasks of chain C_{i-1}. These tasks are organized into the threads of the subcomputation so as to ensure that chain C_i cannot begin executing its trunk tasks until all $\sqrt{T_1}/2$ of the subcomputation's

[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.

[2]Performing the arithmetic reveals that the m chains do not account for all T_1 tasks. The remaining tasks help glue the chains together into a single computation; they are not an important part of this proof and will not be considered further.

threads have been made active, and none of these threads can terminate until chain $\mathcal{C}_{i-1}$ begins executing its blocker tasks. This organization can be accomplished so as to maintain the depth-first property thereby allowing a serial execution to use S_1 space.[3] Nevertheless, in a parallel execution, 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 trunk task of $\mathcal{C}_i$, and let $t_i^{(f)}$ 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 $t_i^{(f)} - t_i^{(s)} \geq \nu$.

Now consider two chains, $\mathcal{C}_i$ and $\mathcal{C}_{i-1}$, and suppose $t_i^{(s)} < t_{i-1}^{(f)}$; this is the scenario we described as using at least $\sqrt{T_1}/2$ space. In this case, we consider the time interval from $t_i^{(s)}$ (inclusive) to $t_{i-1}^{(f)}$ (exclusive) during which we say that chain $\mathcal{C}_i$ is *exposed*, and we let $\tau_i = t_{i-1}^{(f)} - t_i^{(s)}$ denote the amount of time chain $\mathcal{C}_i$ is exposed. If $t_i^{(s)} \geq t_{i-1}^{(f)}$ 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 $T_P(\mathcal{X}) \leq T_1/\rho$—there must be some time step during the execution at which at least $\lceil \frac{3}{4}\rho \rceil - 1$ chains are exposed.

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

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

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

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

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

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

Substituting inequality (2) and inequality (3) back into inequality (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

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

[3]This organization requires that each subcomputation have activation depth at least 3, and therefore, we require $S_1 \geq 4$.

Then the average number of exposed chains per time step is

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

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

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

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

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

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

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 holds even for multithreaded computations with any or all of these restrictions.

5 Scheduling algorithms for strict multithreaded computations

In light of the lower-bound, 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 an execution schedule $\mathcal{X}$ that achieves time $T_P(\mathcal{X}) \leq T_1/P + T_\infty$. 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 valid P-processor execution schedule. In computing such a schedule, the algorithm does not know the entire computation; the computation actually unfolds dynamically during the course of execution, and consequently, the scheduling algorithm must be online. At any given time during the execution, the scheduler has a set of active threads some of which are ready and some of which are stalled. There might be some extra information attached to each thread that the scheduling algorithm can use in deciding which ready threads get executed by which processors, but the scheduler cannot know about the structure of the portion of the computation not yet executed.

To cope with the lower bound from Theorem 4, 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 activation 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.

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 terminates, there is always at least one thread from the subtree rooted at Γ that is ready. This property allows us to derive algorithms to schedule the execution of these computations with efficient use of both time and space.

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

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

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

To prove the space bound, we show that the queue never contains more than P threads (ready or not) that span any activation depth. A thread Γ *spans* an activation depth d, if Γ has activation depth $\mathcal{A}(\Gamma) \ge d$, and either Γ is the root or the parent thread Γ' of Γ has activation depth $\mathcal{A}(\Gamma') < d$. For example, Figure 3 depicts the activation tree corresponding to the computation of Figure 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 active threads that span d at the start of step t. Then the total space $s(t)$ being used at the start of time step t is

$$s(t) = \sum_{d=1}^{S_1} s(t,d) \ . \tag{4}$$

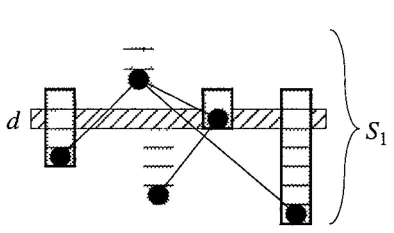

Figure 3: This activation tree corresponds to the example computation of Figure 1. The bold outlined threads span depth d.

By induction on the number of steps, we shall show that for all t, every activation depth d, has $s(t,d) \le P$. With this bound, equation (4) shows that $s(t) \le S_1 P$ for all time t, from which the space bound follows.

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

We can make this algorithm more efficient by reducing the number of accesses to the global queue as follows. Initially, the new 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, terminates, or stalls, the processor will have a thread at the end of the current step. If the thread stalls, 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 terminates, the processor will not have a thread at the end of the step. Lastly, if the thread spawns, the processor returns the parent thread (the one it was working on) to the global queue and keeps the child thread; 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, but it requires access to the global queue only when threads spawn, terminate, or stall.

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

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*, 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; 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 [3].

6 Distributed scheduling algorithms

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

The technique of Karp and Zhang [15] 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 [3]. 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 (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 activation depth d_i of its rth deepest ready thread. In other words, d_i is the activation depth for which processor p_i has fewer than r ready threads deeper than d_i but at least r ready threads at or deeper than d_i. Cutoff depth D is then computed simply by

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

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.

Theorem 7 *Given any number $P \geq 2$ of processors and any strict multithreaded computation with work T_1, computation depth T_∞, and activation depth S_1, the algorithm* LDF($6 \lg P$) *computes a schedule $\mathcal{X}$ that uses space $S_P(\mathcal{X}) = O(S_1 P \lg P)$, and for any $\epsilon > 0$, with probability at least $1-\epsilon$, the schedule uses time $T_P(\mathcal{X}) = O(T_1/P + T_\infty \lg P + \lg(1/\epsilon))$.*

Proof sketch: The space bound follows by an argument similar to the proof of Theorem 5. By induction on the number of iterations, we show that for any activation depth d, the number of active threads that span d is never more than $2rP$. Consider two cases. If depth d begins an iteration with rP

or more active threads spanning it, then by the strictness condition, there must be at least rP ready threads with activation depth at least d. In this case, some processor must have at least r of these ready threads in its local queue, and therefore the cutoff depth for this iteration will be set with $D \geq d$. With the cutoff depth at or below d, no threads shallower than d can be executed during this iteration so the number of active threads spanning d cannot increase. In the other case, if activation depth d begins the iteration with fewer than rP active threads spanning it, then the restriction that no processor performs more than r spawns during an iteration insures that the iteration ends with no more than $2rP$ active threads spanning d. This completes the induction, and equation (4) along with $r = 6 \lg P$ yields the space bound.

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

We now show that with the synchronization parameter set to $r = 6 \lg P$, each iteration fails with probability no more than P^{-5}, independently of the success or failure of any other iteration.[4] Consider an iteration that begins with at least $P \lg P$ ready threads, and suppose that when two threads have the same activation depth, we give each thread a unique identifier to break the tie so we can uniquely identify the $P \lg P$ deepest ready threads. For an iteration that begins with fewer than $P \lg P$ ready threads, we just consider them all. If no local queue contains more than $6 \lg P$ of these deepest ready threads, then the synchronization phase sets the cutoff depth so that all of these deepest threads are at or deeper than the cutoff depth. Therefore, an iteration succeeds if no local queue contains more than $6 \lg P$ of these deepest ready threads. The threads are randomly located among the P local queues, so the number of these deepest threads in any particular queue has a binomial distribution with at most $P \lg P$ trials and success probability $1/P$. One can show that the probability of any particular queue having more than $6 \lg P$ of these deepest ready threads is upper bounded by P^{-6}. Therefore, with probability at least $1 - P^{-5}$, no local queue contains more than $6 \lg P$ of these threads, and the iteration succeeds. That iterations fail independently of each other follows from the fact that each iteration ends by moving each ready thread at or deeper than the cutoff depth to a new randomly chosen queue.

With iterations failing independently of each other, we can bound the total number Y of iterations taken. First, we consider the failed iterations. Let the random variable f denote the number of failed iterations. Since each iteration always results in at least one task being executed, there are at most T_1 iterations. Consequently, f is bounded by a binomial distribution with T_1 trials and success probability P^{-5}, and one can show that for any $\epsilon > 0$, we have $f = O(T_1/(P \lg P) + \log_P(1/\epsilon))$ with probability at least $1 - \epsilon/2$. Now, consider the successful iterations. A successful iteration that begins with at least $P \lg P$ ready threads, executes a task from at least $P \lg P$ of them, and a successful iteration that begins with fewer than $P \lg P$ ready threads, executes a task from every ready thread. Therefore, we can think of each successful iteration as a step in a greedy schedule with $P \lg P$ processors. Then, by Theorem 1, we know that there can be no more than $T_1/(P \lg P) + T_\infty$ successful iterations. Adding together the number of successful iterations and the number of failed iterations, we have that for any $\epsilon > 0$, the total number Y of iterations is

$$Y = O\left(\frac{T_1}{P \lg P} + T_\infty + \log_P(1/\epsilon)\right) \qquad (5)$$

with probability at least $1 - \epsilon/2$.

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 $X = X_1 + X_2 + \cdots + X_Y$. 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 (recall the proof of the space bound)—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, the X_i's are independent. For any $\epsilon > 0$, a moment generating function argument can be used to show that

$$X = O(Y \lg P + \lg(1/\epsilon)) \qquad (6)$$

with probability at least $1 - \epsilon/2$.

Combining equation (5) with equation (6) completes the proof. ■

Corollary 8 *For any number $P \geq 2$ of processors and any strict multithreaded computation with work T_1 and computation depth T_∞, Algorithm* LDF($6 \lg P$) *computes a schedule $\mathcal{X}$ with expected execution time* $\mathrm{E}[T_P(\mathcal{X})] = O(T_1/P + T_\infty \lg P)$. ■

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

The space bound of Theorem 7 is an aggregate bound, but in a distributed memory machine, we may want to bound the space associated with each individual processor's queue. In the LDF algorithm, each active thread is located in the local queue of a processor chosen at random, so we assume that each activation frame is located in the local memory of the same randomly chosen processor as its associated active thread. Since 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

[4]This bound can be tightened, but doing so only improves the constant factors hidden in the big-O time bound.

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 $\mathrm{E}[W] \leq 2rS_1$. Then, using a theorem due to Raghavan [18, Theorem 1], we can show that with probability at least $1 - e^{-2r}$, we have $W \leq 2erS_1$.

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 [3].

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 [20]; dynamic tree-embedding algorithms [2, 17]; and algorithms for backtrack search [14, 19, 25], 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 [25] 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 [6, 12]. 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 [7, 21]. 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. Halstead [12], for example, considered an unfair scheduling policy based on *thread stealing.* In Halstead's thread-stealing strategy, 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 Halstead's unfair scheduling policy, characterizing the performance analytically is difficult.

Thread stealing has also been employed in two parallel chess-playing programs. Zugzwang [10] is a program in which processors steal subcomputations of a chess tree using a parallel alpha-beta search algorithm. StarTech [16] 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 terminates, 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 similar bounds to those for Algorithm LDF. We are currently working on improving these results.

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] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 1990.

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

[3] Robert D. Blumofe. Managing storage for multithreaded computations. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1992. Also: MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-552.

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

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

[6] David E. Culler. Resource management for the tagged token dataflow architecture. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1980.

[7] David E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150, Honolulu, Hawaii, May 1988.

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

[9] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a message-driven processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–196, Pittsburgh, Pennsylvania, June 1987.

[10] R. Feldmann, P. Mysliwietz, and B. Monien. A fully distributed chess program. Technical Report 79, University of Paderborn, Germany, February 1991.

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

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

[13] Robert A. Iannucci. Toward a dataflow / von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, Honolulu, Hawaii, May 1988.

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

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

[16] Bradley Kuszmaul. Private communication, February 1993.

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

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

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

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

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

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

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

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

[25] Yanjun Zhang. *Parallel Algorithms for Combinatorial Search Problems*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, November 1989.