Scheduling Multithreaded Computations by Work Stealing

Robert D. Blumofe
Charles E. Leiserson
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

This paper studies the problem of efficiently scheduling fully strict (i.e., well-structured) multithreaded computations on parallel computers. A popular and practical method of scheduling this kind of dynamic MIMD-style computation is "work stealing," in which processors needing work steal computational threads from other processors. In this paper, we give the first provably good work-stealing scheduler for multithreaded computations with dependencies.

Specifically, our analysis shows that the expected time T_P to execute a fully strict computation on P processors using our work-stealing scheduler is $T_P = O(T_1/P + T_\infty)$, where T_1 is the minimum serial execution time of the multithreaded computation and T_∞ is the minimum execution time with an infinite number of processors. Moreover, the space S_P required by the execution satisfies $S_P \leq S_1 P$. We also show that the expected total communication of the algorithm is at most $O(T_\infty S_{\max} P)$, where $S_{\max}$ is the size of the largest activation record of any thread, thereby justifying the folk wisdom that work-stealing schedulers are more communication efficient than their work-sharing counterparts. All three of these bounds are existentially optimal to within a constant factor.

1 Introduction

For efficient execution of a dynamically growing "multithreaded" computation on a MIMD-style parallel computer, a scheduling algorithm must ensure that enough threads remain concurrently active to keep the processors busy. Simultaneously, it should ensure that the number of concurrently active threads remains within reasonable limits so that memory requirements can be bounded. Moreover, the scheduler should also try to maintain related threads on the same processor, if possible, so that communication between them can be minimized. Needless to say, achieving all these goals simultaneously can be difficult.

Two scheduling paradigms have arisen to address the problem of scheduling multithreaded computations: *work sharing* and *work stealing*. In work sharing, whenever a processor generates new threads, the scheduler attempts to migrate some of them to other processors in hopes of distributing the work to underutilized processors. In work stealing, however, underutilized processors take the initiative: they attempt to "steal" threads from other processors. Intuitively, the migration of threads occurs less frequently with work stealing than with work sharing, since if all processors have plenty of work to do, no threads are migrated by a work-stealing scheduler, but threads are always migrated by a work-sharing scheduler.

The work-stealing idea dates back at least as far as Burton and Sleep's research [7] on parallel execution of functional programs and Halstead's implementation of Multilisp [14]. Since then, many researchers have implemented variants on this strategy [4, 9, 10, 13, 16, 18, 24]. Rudolph, Slivkin-Allalouf, and Upfal [22] analyzed a randomized work-stealing strategy for load balancing independent jobs on a parallel computer, and Karp and Zhang [15] analyzed a randomized work-stealing strategy for parallel backtrack search. Recently, Zhang and Ortynski [26] have obtained good bounds on the communication requirements of the randomized parallel backtrack search algorithm presented in [15].

In this paper, we present and analyze a work-stealing algorithm for scheduling "fully strict" (well-structured) multithreaded computations. This class of computations encompasses both backtrack search computations [15, 26] and divide-and-conquer computations [25], as well as dataflow computations [1] in which threads may stall due to a data dependency. We analyze our algorithms in a stringent atomic access

This research was supported in part by the Advanced Research Projects Agency under Grant N00014-91-J-1698. Robert D. Blumofe is supported in part by an ARPA High-Performance Computing Graduate Fellowship.

model similar to the atomic message-passing model of [17] in which concurrent accesses to the same data structure are serially queued by an adversary.

Our main contribution is a randomized work-stealing scheduling algorithm for fully strict multithreaded computations which is provably efficient in terms of time, space, and communication. The bounds on space and time are better than previous bounds for work-sharing schedulers [3], and the work-stealing scheduler is much simpler and eminently practical. Part of this improvement is due to our focusing on fully strict computations, as compared to the (general) strict computations studied in [3]. Moreover, we are also able to provide a bound on the communication of fully strict computations which is existentially tight to within a constant factor, meeting the lower bound of Wu and Kung [25] for communication in parallel divide-and-conquer. In contrast, work-sharing schedulers have near worst-case behavior for communication. Thus, our results bolster the folk wisdom that work stealing is superior to work sharing.

Others have studied and continue to study the problem of efficiently managing the space requirements of parallel computations. Culler and Arvind [8] and Ruggiero and Sargeant [23] give heuristics for limiting the space required by dataflow programs. Burton [6] shows how to limit space in certain parallel computations without causing deadlock.

The remainder of this paper is organized as follows. In Section 2 we review the graph-theoretic model of multithreaded computations introduced in [3], which provides a theoretical basis for analyzing schedulers. Section 3 gives a simple scheduling algorithm which uses a central queue. This "busy-leaves" algorithm forms the basis for our randomized work-stealing algorithm, which we present in Section 4. In Section 5 we introduce the atomic-access model that we use to analyze execution time and communication costs for the work-stealing algorithm, and we present and analyze a combinatorial "balls and bins" game that we use to derive a bound on the contention that arises in random work stealing. We then use this bound along with a delay-sequence argument [21] in Section 6 to analyze the execution time and communication cost of the work-stealing algorithm. We make some concluding remarks in Section 7.

2 A model for multithreaded computation

This section reprises the graph-theoretic model of multithreaded computation introduced in [3]. We also define what it means for computations to be "fully strict." We conclude with a statement of the greedy-scheduling theorem, which is an adaptation of theorems by Brent [5] and Graham [11, 12] on dag scheduling.

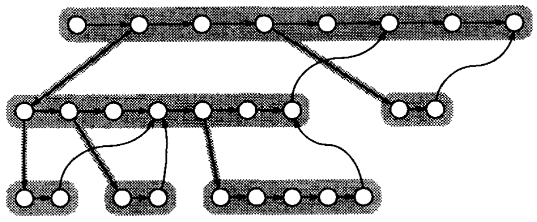
Figure 1: A multithreaded computation.

A multithreaded computation is composed of a set of threads, each of which is a sequential ordering of unit-time 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 executes at most one task.

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, and a thread can spawn as many children as it desires. In this way, threads are organized into an *activation tree* hierarchy as indicated in Figure 1 by the shaded edges, called *spawn* edges. Each spawn edge goes from a specific task, the task that actually does the spawn operation, in the parent thread to the first task of the child thread. When a thread executes its last task, it *dies*.

For an execution schedule to be valid, the task execution order must obey the ordering constraints given by the directed 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 which 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. We make the reasonable assumption that each task has at most a constant number of data-dependency edges incident on it. 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 ***satisfied***, which ***enables*** the consuming thread to proceed with its execution—the thread becomes ***ready***.

To summarize, a multithreaded computation can be viewed as a bounded-degree directed graph of tasks connected by continue, spawn, and data-dependency edges. We assume that this graph is a dag (directed acyclic graph), so that valid execution schedules do exist. The tasks are connected by continue edges into threads, and the threads form an activation tree with the spawn edges.

Because multithreaded computations with arbitrary data dependencies can be impossible to schedule efficiently [3], we study subclasses of general multithreaded computations in which the kinds of data-dependencies that can occur are restricted. A ***strict*** multithreaded computation is one in which all data-dependency edges from a thread go to an ancestor of the thread in the activation tree. This restriction essentially means that a subroutine cannot be invoked before all of its arguments are available, although the arguments can be garnered in parallel. A ***fully strict*** computation is one in which all data-dependency edges from a thread go to the thread's parent. A fully strict computation is, in a sense, a "well-structured" computation, in that all data dependencies from a subtree emanate from the subtree's root. Any multithreaded computation that can be executed in a depth-first manner on a single processor can be made either strict or fully strict by altering the dependency structure, possibly affecting the achievable parallelism, but not affecting the semantics of the computation.

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 ***alive***. Recall that at any time, a live thread can be either stalled or ready, but even if it stalls, its activation frame remains allocated. The thread remains alive until it dies; at that point its frame can be deallocated.

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 technically not necessary, it gives the execution a natural structure, and it will simplify our analyses of space utilization. At a given time t during the execution of a computation, the ***activation subtree*** at time t is the portion of the activation tree consisting of just those threads that are alive at time t. We also assume that the frames hold all the values used by the computation, and thus, 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 threads in the activation subtree at that time, and the total space used in executing a computation is the maximum such value over the course of the execution.

To quantify the space used by a given execution schedule of a computation, 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 of its threads. We shall denote by S_1 the minimum amount of space possible for any 1-processor execution of a strict computation, which is equal to the activation depth of the computation. Let S_P denote the space used by a P-processor execution schedule of a strict multithreaded computation. We shall be interested in those execution schedules that exhibit only linear expansion of space, that is, $S_P = O(S_1 P)$.

To quantify execution time in the multithreaded model, let T_P denote the time used by a P-processor execution schedule. We define the ***dag depth*** of a task to be the length of a longest path that terminates at the task. We define the ***dag depth*** of the entire multithreaded computation as the maximum dag depth of any task in the computation. We denote the dag depth of a computation by T_∞, because with an infinite number of processors, progress can always be made along the "critical path" of the computation. Notice that $T_P \geq T_\infty$, since the tasks along any path must be executed in a serial order. We also define the ***work*** of a multithreaded computation to be the number of tasks in the computation. The work is denoted T_1, because it is the minimum time it takes to execute the computation with 1 processor. Notice that

$T_P \geq T_1/P$, because P processors can execute only P tasks per time step.

Early work on dag scheduling by Brent [5] and Graham [11, 12] shows that there exist P-processor execution schedules with $T_P \leq T_1/P+T_\infty$. The following theorem, proved in [3], extends these results 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 (The greedy-scheduling theorem) *For any multithreaded computation with work T_1 and dag depth T_∞, for any number P of processors, any greedy execution schedule achieves $T_P \leq T_1/P + T_\infty$.* ∎

Generally, we are interested in schedules that achieve linear speedup, that is $T_P = O(T_1/P)$. For a greedy schedule, linear speedup occurs when the *average available parallelism*, which we define to be T_1/T_∞, satisfies $T_1/T_\infty = \Omega(P)$.

3 The busy-leaves algorithm

This section presents and analyzes Algorithm BL (for "busy leaves"), which schedules strict multithreaded computations on a parallel machine with P processors. We show that the time to execute a strict multithreaded computation with work T_1, dag depth T_∞, and activation depth S_1 on a P-processor machine is $O(T_1/P + T_\infty)$, excluding scheduling overheads, thus matching the upper bound from Theorem 1. Moreover, the scheduling algorithm displays only linear expansion of space.

In Algorithm BL, all living threads are maintained in a single thread pool which is uniformly available to all P processors. When spawns occur, new threads are added to this global pool, and when a processor needs work, it removes a ready thread from the thread pool. In our analysis, we ignore the effects of processors contending for access to the pool. Specifically, we assume that each processor can add threads to the thread pool and delete threads from it in unit time, regardless of what the other processors are doing. Under this assumption we show that Algorithm BL can execute a strict multithreaded computation with linear expansion of space and linear speedup, provided that the average available parallelism is $\Omega(P)$. This algorithm is likely to perform well for small symmetric multiprocessors, but contention at the thread pool limits its ability to scale to large-scale systems. We overcome this limit on scalability in Section 4, where we present our work-stealing scheduler, which is based on Algorithm BL.

Algorithm BL operates as follows. Whenever a processor has no thread to work on, it removes any ready thread, call it A, from the thread pool and begins work on it. The processor continues executing A's tasks until the thread either spawns, stalls, or dies.

1. If the thread A spawns a child B, then A is returned to the thread pool, and the processor commences work on B.
2. If the thread A stalls, then A is returned to the thread pool, and the processor obtains new work by removing any ready thread from the pool.
3. If the thread A dies, then the processor checks to see if A's parent thread B currently has any living children. If B has no live children and no other processor is working on B, then the processor takes B from the pool and commences working on it. Otherwise, the processor takes any ready thread from the pool.

As the proof of the following theorem shows, Algorithm BL guarantees that all leaves in the activation subtree have processors working on them, that is, all leaves are busy.

Theorem 2 *For any strict multithreaded computation with work T_1, dag depth T_∞, and activation depth S_1, and for any number P of processors, Algorithm BL computes a P-processor execution schedule whose execution time T_P satisfies $T_P \leq T_1/P + T_\infty$ and whose space S_P satisfies $S_P \leq S_1 P$.*

Proof: The time bound follows from Theorem 1. The space bound is a consequence of the fact that for strict computations, at any time during the execution, every leaf in the activation subtree has a processor working on it. This *busy-leaves property* can be proved by induction on execution time. The busy-leaves property implies that at all times, the activation subtree has at most P leaves. For each such leaf, the space used by it and all of its ancestors is at most S_1, and therefore, the total size of the activation subtree is at most $S_1 P$. ∎

4 A randomized work-stealing algorithm for fully strict computations

In this section, we present Algorithm WS (for "work stealing"), a randomized work-stealing algorithm for scheduling fully strict multithreaded computations on a parallel computer. Also, we show that for fully strict computations, this algorithms causes at most a linear expansion of space, and we present an important structural lemma which will be used in Section 6 to analyze the execution time of this algorithm.

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

In general, a processor obtains work by removing the thread at the bottom of its ready deque. It starts working on the thread, call it A, and continues executing A's tasks until A enables a stalled thread, spawns, dies, or stalls, in which case, it performs one of the following actions.

1. If the thread A enables a stalled thread B (A's parent), the now-ready thread B is placed in the (empty) ready deque of A's processor.
2. If the thread A spawns a child B, then A is placed on the bottom of the ready deque, and the processor commences work on B.
3. If the thread A dies or stalls, its processor checks the ready deque. If the deque contains any threads, then the processor removes and begins work on the bottommost thread. If the ready deque is empty, however, the processor begins work stealing: it steals the topmost thread from the ready deque of a randomly chosen processor and begins work on it. (This work-stealing strategy is elaborated below.)

A thread can simultaneously enable a stalled thread and die, in which case we first perform the action for enabling and then the one for dying. Algorithm WS begins with all ready deques empty. The root thread of the multithreaded computation is placed in the ready deque of one processor, while the other processors start work stealing, as in step 3 above.

When a processor begins work stealing, it operates as follows. The processor becomes a ***thief*** and attempts to steal work from a ***victim*** processor chosen uniformly at random. The thief queries the ready deque of the victim, and if it is nonempty, the thief removes and begins work on the top thread. If the victim's ready deque is empty, however, the thief tries again, picking another victim at random.

We now bound the space used by Algorithm WS.

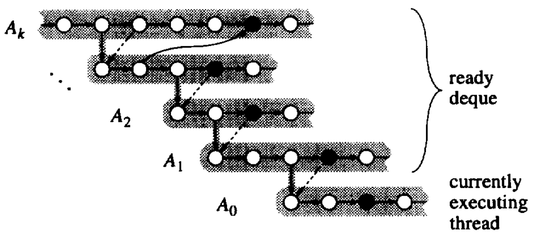

Figure 2: The structure of a processor's ready deque. The black task in each thread indicates the thread's currently ready task. The dashed edges are the "deque edges" introduced in Section 6.

Theorem 3 *For any fully strict multithreaded computation with activation depth S_1, Algorithm WS run on a computer with P processors uses at most S_1P space.*

Proof: Like Algorithm BL, Algorithm WS maintains the busy-leaves property: at all times during the execution, every leaf in the current activation subtree has a processor working on it. ■

We conclude this section with an important lemma on the structure of threads in the ready deque of any processor during the execution of a fully strict computation scheduled by Algorithm WS. This lemma is used in the next section to analyze execution time. Figure 2 illustrates this lemma.

Lemma 4 *During the execution of any fully strict multithreaded computation by Algorithm WS, consider any processor p and any given time during which p is working on a thread. Let A_0 be the thread that p is working on, let k be the number of threads in p's ready deque, and let $A_1, A_2, \ldots, A_k$ denote the threads in p's ready deque ordered from bottom to top, so that A_1 is the bottommost and A_k is the topmost. Then the threads in p's ready deque satisfy the following properties:*

i. For $i = 1, 2, \ldots, k$, thread A_{i-1} is a child of A_i in the current activation subtree.

ii. For $i = 1, 2, \ldots, k-1$, thread A_i has not been worked on since it spawned A_{i-1}.

Proof: The proof is a straightforward induction on execution time. The only interesting issue is how it happens that thread A_k might have been worked on since it spawned A_{k-1}, since Condition ii excludes A_k. This situation arises when A_k is stolen from processor p and then stalls on its new processor. Later, A_k is reenabled by A_{k-1} and brought back to processor p's ready deque. The key observation is that when A_k is reenabled, processor p's ready deque is empty and p is working on A_{k-1}. The other threads $A_{k-2}, A_{k-3}, \ldots, A_0$ shown in Figure 2 were spawned after A_k was reenabled. ∎

5 The atomic-access model and the recycling game

This section presents the "atomic-access" model that we use to analyze contention during the execution of a multithreaded computation by Algorithm WS. We introduce a combinatorial "balls and bins" game, which we use to bound the total amount of delay incurred by random, asynchronous accesses in this model. We shall use the results of this section in Section 6, where we analyze Algorithm WS.

The *atomic-access model* is the machine model we use to analyze Algorithm WS. We assume that the machine is an asynchronous parallel computer with P processors, and its memory can be either distributed or shared. Our analysis assumes that concurrent accesses to the same data structure are serially queued by an adversary, as in the atomic message-passing model of [17]. This assumption is more stringent than that in the model of Karp and Zhang [15]. They assume that if concurrent steal requests are made to a queue, in one time step, one request is satisfied and all the others are denied. In the atomic-access model, we also assume that one request is satisfied, but the others are queued by an adversary, rather than being denied. Moreover, from the collection of waiting requests for a given deque, the adversary gets to choose which is serviced and which continue to wait. The only constraint on the adversary is that if there is at least one request for a deque, then the adversary cannot choose that none be serviced.

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

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

1. He chooses some of the balls in the reservoir (possibly all and possibly none), and then for each of these balls, he removes it from the reservoir, selects one of the P bins uniformly and independently at random, and tosses the ball into it.
2. He inspects each of the P bins in turn, and for each bin that contains at least one ball, he removes any one of the balls in the bin and returns it to the reservoir.

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

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

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

Lemma 5 *For any $\epsilon > 0$, with probability at least $1 - \epsilon$, the total delay in the (P, M)-recycling game is $O(M + P \lg P + P \lg(1/\epsilon))$. The expected total delay is at most M. In other words, the total delay incurred by M random requests made by P processors in the atomic-access model is $O(M + P \lg P + P \lg(1/\epsilon))$[1] with probability at least $1 - \epsilon$, and the expected total delay is at most M.*

Proof: We first make the observation that the strategy by which the adversary chooses a ball from each bin is immaterial, and thus, we can assume that balls

[1]Greg Plaxton of University of Texas, Austin has recently improved this bound to $O(M)$ for the case when $1/\epsilon$ is at most polynomial in M and P [20].

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

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

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

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

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

We say that the ith cycle of ball 1 is ***delayed*** by another ball r if the ith toss of ball 1 places it in some bin k, and ball r is removed from bin k during the ith cycle of ball 1. Since the adversary follows the FIFO rule, it follows that the ith cycle of ball 1 can be delayed by another ball r either once or not at all. Consequently, we can decompose each random variable d_i into a sum $d_i = x_{i2} + x_{i3} + \cdots + x_{im}$ of indicator random variables where

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

Thus, we have

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

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

$$\Pr\left\{ \bigwedge_{(i,r)\in S} (x_{ir} = 1) \right\} \le P^{-|S|} \ . \tag{3}$$

The crux of proving the claim is to show that

$$\Pr\left\{ x_{ir} = 1 \,\middle|\, \bigwedge_{(i',r')\in S'} (x_{ir} = 1) \right\} \le 1/P \ , \tag{4}$$

where $S' = S - \{(i, r)\}$, whence the claim Inequality (3) follows from Bayes's Theorem.

We can derive Inequality (4) from a careful analysis of dependencies. Because the adversary follows the FIFO rule, we have that $x_{ir} = 1$ only if, when the adversary executes the ith toss of ball 1, it falls into whatever bin contains ball r, if any. *A priori*, this event happens with probability either $1/P$ or 0, and hence, with probability at most $1/P$. Conditioning on any collection of events relating which balls delay this or other cycles of ball 1 cannot increase this probability, as we now argue in two cases. In the first case, the indicator random variables $x_{i'r'}$, where $i' \ne i$, tell whether other cycles of ball 1 are delayed. This information tells nothing about where the ith toss of ball 1 goes. Therefore, these random variables are independent of x_{ir}, and thus, the probability $1/P$ upper bound is not affected. In the second case, the indicator random variables $x_{ir'}$ tell whether the ith toss of ball 1 goes to the bin containing ball r', but this information tells us nothing about whether it goes to the bin containing ball r, because the indicator random variables tell us nothing to relate where ball r and ball r' are located. Moreover, no "collusion" among the indicator random variables provides any more information, and thus Inequality (4) holds.

Equation (2) shows that the delay δ encountered by ball 1 throughout all of its cycles can be expresses as a sum of $m(P-1)$ indicator random variables. In order for δ to equal or exceed a given value Δ, there must be some set containing Δ of these indicator random variables, each of which must be 1. For any specific such set, Inequality (3) says that the probability is at most $P^{-\Delta}$ that all random variables in the set are 1. Since there are $\binom{m(P-1)}{\Delta} \le (emP/\Delta)^{\Delta}$ such sets, we have

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

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

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

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

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

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

6 Analysis of the work-stealing algorithm

In this section, we analyze the time and communication cost of executing a fully strict multithreaded computation with Algorithm WS. For fully strict computations on P processors, we show that the expected running time, including scheduling overhead, is $O(T_1/P + T_\infty)$. Moreover, for any $\epsilon > 0$, with probability at least $1-\epsilon$, the execution time on P processors is $T_P = O(T_1/P + T_\infty + \lg P + \lg(1/\epsilon))$. We also show that the expected total communication during the execution of a computation is $O(PT_\infty S_{\max})$, where $S_{\max}$ is the largest size of any activation record.

Unlike in Algorithm BL, the "ready pool" in Algorithm WS is distributed, and so there is no contention at a centralized data structure. Nevertheless, it is still possible for contention to arise when several thieves happen to descend on the same victim simultaneously. In this case, as we have indicated in Section 5, we make the conservative assumption that an adversary serially queues the work-stealing requests as in the atomic message-passing model of [17].

To analyze the running time of Algorithm WS executing a fully strict multithreaded computation with work T_1 and dag depth T_∞ on a computer with P processors, we use an accounting argument. At each step of the algorithm, we collect P dollars, one from each processor. At each step, each processor places its dollar in one of three buckets according to its actions at that step. If the processor executes a task at the step, then it places its dollar into the *Work* bucket. If the processor initiates a steal attempt at the step, then it places its dollar into the *Steal* bucket. And if the processor merely waits for a queued steal request at the step, then it places its dollar into the *Wait* bucket. We shall derive the running time bound by bounding the number of dollars in each bucket at the end of the execution, summing these three bounds, and then dividing by P.

We first bound the total number of dollars in the *Work* bucket.

Lemma 6 *The execution of a fully strict multithreaded computation with work T_1 by Algorithm WS on a computer with P processors terminates with exactly T_1 dollars in the Work bucket.*

Proof: A processor places a dollar in the *Work* bucket only when it executes a task. Thus, since there are T_1 tasks in the computation, the execution ends with exactly T_1 dollars in the *Work* bucket. ∎

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

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

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

The deque edges are the key to defining critical tasks. At any time during the execution, we say that a task v is *critical* if every task that precedes v (either directly or indirectly) in D' has been executed, that is, if for every task w such that there is a directed path from w to v in D', task w has been executed. A critical task must be ready, since D' contains every edge of D, but a ready task may or may not be critical. Intuitively, the structural properties of a ready deque enumerated in Lemma 4 guarantee that if a thread is deep in a ready deque, then it cannot be critical, because the predecessor of the thread's current task across the deque edge has not yet been executed.

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

We now formalize our definition of a delay sequence.

Definition 7 *A delay sequence is a 3-tuple (U,R,Π) satisfying the following conditions:*

- *$U=(u_1,u_2,\ldots,u_L)$ is a maximal directed path in D'. That is, for $i=1,2,\ldots,L-1$, the edge (u_i,u_{i+1}) belongs to D', task u_1 has no incoming edges in D' (task u_1 must be the first task of the root thread), and task u_L has no outgoing edges in D' (task u_L must be the last task of the root thread).*
- *R is a positive integer.*
- *$\Pi=(\pi_1,\pi_2,\ldots,\pi_L)$ is a partition of the integer R.*

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

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

Lemma 8 *Consider the execution of a fully strict multithreaded computation with dag depth T_∞ by Algorithm WS on a computer with P processors. If at least $4P(2T_\infty+R)$ steal attempts occur during the execution, then some (U,R,Π) delay sequence must occur.*

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

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

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

Lemma 9 *At all times during the execution of a fully strict multithreaded computation by Algorithm WS, each critical task must be the ready task of a thread that is one of the top 2 in its processor's ready deque.*

Proof: Consider any time step, and let u_0 be the critical task of a thread B_0. Since u_0 is critical, B_0 must be ready, and therefore, B_0 must be in the ready deque of some processor p. If B_0 is not one of the top 2 threads in p's ready deque, then Lemma 4 guarantees that each of the at least 2 threads above B_0 in p's ready deque is an ancestor of B_0. Let $B_1, B_2, \ldots, B_k$ denote B_0's ancestor threads, where B_1 is the parent of B_0 and B_k is the root thread. Further, for $i = 1, 2, \ldots, k$, let u_i denote the task of thread B_i that spawned thread B_{i-1}, and let w_i denote u_i's successor task in thread B_i. In the dag D', we have deque edges (w_i, u_{i-1}) for $i = 1, 2, \ldots, k$. Consequently, since u_0 is critical, for $i = 1, 2, \ldots, k$, each task w_i must have been executed, since it is a predecessor of u_0 in D'. Moreover, because each w_i is the successor of the spawn task u_i in thread B_i, each thread B_i for $i = 1, 2, \ldots, k$ must have been worked on since the time that it spawned thread B_{i-1}. But Lemma 4 guarantees that only the topmost thread in p's ready deque can have this property. Thus, B_1 is the only thread that can possibly be above B_0 in p's ready deque. ■

Lemma 10 *Consider the execution of any fully strict multithreaded computation by Algorithm WS on a parallel computer with $P \geq 2$ processors. For any task v and any set of $r \geq 4$ work-steal rounds, the probability that the r rounds occur while the task is critical is at most the probability that only 0 or 1 of the steal attempts initiated in the r rounds choose v's processor, which is at most e^{-2r}*

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

We shall first show that of these $(3r-1)P$ steal attempts initiated while task v is critical and at least 2 time steps before v is executed, at most 1 of them can choose processor p as its target, for otherwise, v would be executed at or before t_b. Recall from Lemma 9 that task v is the ready task of a thread B, which must be among the top 2 threads in p's ready deque as long as v is critical.

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

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

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

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

for $r \geq 4$. ■

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

Lemma 11 *Consider the execution of any fully strict multithreaded computation with dag depth T_∞ by Algorithm WS on a parallel computer with P processors. For any $\epsilon > 0$, with probability at least $1 - \epsilon$, at most $O(P(T_\infty + \lg(1/\epsilon)))$ work-steal attempts occur. The expected number of steal attempts is $O(PT_\infty)$. In other words, with probability at least $1-\epsilon$, the execution terminates with at most $O(P(T_\infty + \lg(1/\epsilon)))$ dollars in the Steal bucket, and the expected number of dollars in this bucket is $O(PT_\infty)$.*

Proof: From Lemma 8, we know that if at least $4P(2T_\infty + R)$ steal attempts occur, then some delay sequence (U, R, Π) must occur. Now, consider a particular delay sequence (U, R, Π) having $U = (u_1, u_2, \ldots, u_L)$ and $\Pi = (\pi_1, \pi_2, \ldots, \pi_L)$ where $\pi_1 + \pi_2 + \cdots + \pi_L = R$ and $L \leq 2T_\infty$. We shall compute the probability that (U, R, Π) occurs.

Such a sequence occurs if, for each $i = 1, 2, \ldots, L$, at least π_i work-steal rounds occur while task u_i is critical. From Lemma 10, we know that the probability of at least π_i rounds occurring while a given task u_i is critical is at most the probability that only 0 or 1 steal attempts initiated in the π_i rounds choose v's processor, which is at most $e^{-2\pi_i}$ provided $\pi_i \geq 4$. (For those values of i with $\pi < 4$, we use 1 as an upper bound on this probability.) Moreover, since the targets of the work-steal attempts in the π_i rounds are chosen independently from the targets chosen in other rounds, we can bound the probability of the particular delay sequence (U, R, Π) occurring as follows:

$$
\begin{aligned}
\Pr\{(U,R,\Pi) \text{ occurs}\} &= \prod_{1 \leq i \leq L} \Pr\{\pi_i \text{ rounds occur while } u_i \text{ is critical}\} \\
&\leq \prod_{\substack{1 \leq i \leq L \\ \pi_i \geq 4}} e^{-2\pi_i} \\
&= \exp\left[-2 \sum_{\substack{1 \leq i \leq L \\ \pi_i \geq 4}} \pi_i\right] \\
&= \exp\left[-2\left(\sum_{1 \leq i \leq L} \pi_i - \sum_{\substack{1 \leq i \leq L \\ \pi_i < 4}} \pi_i\right)\right] \\
&\leq e^{-2(R-3L)} .
\end{aligned}
$$

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

$$
(d+1)^{2T_\infty} \binom{2T_\infty + R}{R} e^{-2R+12T_\infty} , \tag{5}
$$

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

With bounds on the number of dollars in the *Work* and *Steal* buckets, we now state the theorem that bounds the total execution time for a fully strict multithreaded computation by Algorithm WS, and we complete the proof by bounding the number of dollars in the *Wait* bucket.

Theorem 12 *Consider the execution of any fully strict multithreaded computation with dag depth T_∞ and work T_1 by Algorithm WS on a parallel computer with P processors. The expected running time, including scheduling overhead, is $O(T_1/P + T_\infty)$. Moreover, for any $\epsilon > 0$, with probability at least $1-\epsilon$, the execution time on P processors is $T_P = O(T_1/P + T_\infty + \lg P + \lg(1/\epsilon))$.*[2]

Proof: Lemmas 6 and 11 bound the dollars in the *Work* and *Steal* buckets, so we now must bound the dollars in the *Wait* bucket. This bound is given by Lemma 5 which bounds the total delay—that is, the total dollars in the *Wait* bucket—as a function of the number M of steal attempts—that is, the total dollars in the *Steal* bucket. This lemma says that for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the number of dollars in the *Wait* bucket is at most a constant

[2]With Plaxton's bound [20] for Lemma 5, this bound becomes $T_P = O(T_1/P + T_\infty)$, whenever $1/\epsilon$ is at most polynomial in M and P.

times the number of dollars in the *Steal* bucket plus $O(P \lg P + P \lg(1/\epsilon))$, and the expected number of dollars in the *Wait* bucket is at most the number in the *Steal* bucket.

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

The next theorem bounds the total amount of communication that a multithreaded computation executed by Algorithm WS performs in a distributed model. The analysis makes the assumption that the size of a parent thread's activation record is at least proportional to the maximum number of data-dependency edges entering the parent from any of its children. It also assumes that at most a constant number of messages need be communicated along a dependency edge to resolve the dependency.

Theorem 13 *Consider the execution of any fully strict multithreaded computation with dag depth T_∞ by Algorithm WS on a parallel computer with P processors. Then, the total number of bytes communicated has expectation $O(PT_\infty S_{\max})$, where $S_{\max}$ is the size in bytes of the largest activation record in the computation. Moreover, for any $\epsilon > 0$, the probability is at least $1 - \epsilon$ that the total communication incurred is $O(P(T_\infty + \lg(1/\epsilon))S_{\max})$.*

Proof: We prove the bound for the expectation. The high-probability bound is analogous. By our bucketing argument, the expected number of steal attempts is at most $O(PT_\infty)$, and when a thread is stolen, the communication incurred is at most $S_{\max}$. Thus, the expected total communication for stealing threads is $O(PT_\infty S_{\max})$. Communication also occurs whenever a dependency edge enters a parent thread from one of its children and the parent has been stolen. But since each dependency edge accounts for at most a constant number of messages, we can amortize the communications cost of resolving all dependencies to the parent by the cost $S_{\max}$ of stealing the parent. Finally, we can have communication when a child thread enables its parent and puts the parent into the child's processor's ready deque. This event can happen only once for each time the parent is stolen, however, and thus, once again the cost can be amortized by the number of steal attempts. Thus, the expected total communication cost is $O(PT_\infty S_{\max})$. ∎

The communication bounds in this theorem are existentially tight, in that there exist fully strict computations that require $\Omega(PT_\infty S_{\max})$ total communication for any execution schedule. This result follows directly from a theorem of Wu and Kung [25], who showed that divide-and-conquer computations—a special case of fully strict computations—require this much communication.

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

7 Conclusion

We are currently trying to extend our theoretical work in several ways. In this paper, we bounded the total amount of space required by a multithreaded computation, but we did not guarantee that any individual processor does not run out of space. In recent work, we have been able to modify Algorithm WS to achieve per-processor space bounds, but the modified algorithm is messy and impractical. We have also looked at designing a provably good work-stealing scheduler for (general) strict computations, but our efforts here have not been as fruitful, because the structure described by Lemma 4 appears too fragile to extend easily beyond fully strict computations. More ambitiously, we are exploring whether provably good schedulers exist for computations involving specific, commonly occurring instances of nonstrictness, such as on-going producer-consumer relationships and speculative computations.

How practical are the methods analyzed in this paper? We have been actively engaged in building a C-based language called "Cilk" for programming multithreaded computations [2]. We currently have preliminary versions of the system that run on the Connection Machine CM-5, the Phish runtime system for networks of Unix workstations [4], and the Sparcstation 10 symmetric multiprocessor. Cilk is derived from the PCM "parallel continuation machine" system [13], which was itself inspired in part by the research reported here. The per-processor overhead of the Cilk implementation, compared with a native C implementation, is typically at most 15 percent on various applications that we have programmed. To date, our applications include a protein-folding program [19], which was the first program to find the number of hamiltonian paths in a $4 \times 4 \times 3$ grid, and a parallel chess-playing program ⋆Socrates, which won third prize at the 1994 ACM International Computer Chess Championship.

Acknowledgments

Thanks to Bruce Maggs of Carnegie Mellon, who outlined the strategy in Section 6 for using a delay sequence argument to prove the time bounds on Algorithm WS, which improved our previous bounds. Thanks to Greg Plaxton of University of Texas, Austin for technical comments on our probabilistic analyses. Thanks to Yanjun Zhang of Southern Methodist University and Warren Burton of Simon Fraser University for comments that improved the clarity of our paper. Thanks also to Arvind, Michael Halbherr, Chris Joerg, Bradley Kuszmaul, Keith Randall, and Yuli Zhou of MIT for helpful discussions.

References

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

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Phil Lisiecki, Keith H. Randall, Andy Shaw, and Yuli Zhou. Cilk 1.0 reference manual. MIT Technical Report, to appear, 1994.

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994.

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

[18] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991. Also: MIT Laboratory for Computer Science Technical Report MIT/LCS/TM-449.

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

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

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

[22] 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.

[23] 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.

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

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

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