Scheduling Multithreaded Computations by Work Stealing

Robert D. Blumofe
The University of Texas at Austin
Charles E. Leiserson
MIT Laboratory for Computer Science

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 to execute a fully strict computation on P processors using our work-stealing scheduler is $T_1/P + O(T_\infty)$, 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 required by the execution is at most $S_1 P$, where S_1 is the minimum serial space requirement. We also show that the expected total communication of the algorithm is at most $O(PT_\infty(1 + n_d)S_{\max})$, where $S_{\max}$ is the size of the largest activation record of any thread and n_d is the maximum number of times that any thread synchronizes with its parent. This communication bound justifies 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 are active concurrently to keep the processors busy. Simultaneously, it should ensure that the number of concurrently active threads remains within reasonable limits so that memory requirements are not unduly large. Moreover, the scheduler should also try to maintain related threads

This research was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985. This research was done while Robert D. Blumofe was at the MIT Laboratory for Computer Science and was supported in part by an ARPA High-Performance Computing Graduate Fellowship.

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 when all processors have 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 on parallel execution of functional programs [16] and Halstead's implementation of Multilisp [30]. These authors point out the heuristic benefits of work stealing with regards to space and communication. Since then, many researchers have implemented variants on this strategy [11, 21, 23, 29, 34, 37, 46]. Rudolph, Slivkin-Allalouf, and Upfal [43] analyzed a randomized work-stealing strategy for load balancing independent jobs on a parallel computer, and Karp and Zhang [33] analyzed a randomized work-stealing strategy for parallel backtrack search. Recently, Zhang and Ortynski [48] have obtained good bounds on the communication requirements of this algorithm.

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 [33, 48] and divide-and-conquer computations [47], as well as dataflow computations [2] in which threads may stall due to a data dependency. We analyze our algorithms in a stringent atomic-access model similar to the atomic message-passing model of [36] 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. We prove that the expected time to execute a fully strict computation on P processors using our work-stealing scheduler is $T_1/P + O(T_\infty)$, 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. In addition, the space required by the execution is at most $S_1 P$, where S_1 is the minimum serial space requirement. These bounds are better than previous bounds for work-sharing schedulers [10], 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 [10]. We also prove that the expected total communication of the execution is at most $O(PT_\infty(1 + n_d)S_{\max})$, where $S_{\max}$ is the size of the largest activation record of any thread and n_d is the maximum number of times that any thread synchronizes with its parent. This bound is existentially tight to within a constant factor, meeting the lower bound of Wu and Kung [47] for communication in parallel divide-and-conquer. In contrast, work-sharing schedulers have nearly 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 [19] and Ruggiero and Sargeant

[44] give heuristics for limiting the space required by dataflow programs. Burton [14] shows how to limit space in certain parallel computations without causing deadlock. More recently, Burton [15] has developed and analyzed a scheduling algorithm with provably good time and space bounds. Blelloch, Gibbons, Matias, and Narlikar [3, 4] have also recently developed and analyzed scheduling algorithms with provably good time and space bounds. It is not yet clear whether any of these algorithms are as practical as work stealing.

The remainder of this paper is organized as follows. In Section 2 we review the graph-theoretic model of multithreaded computations introduced in [10], 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 [41] in Section 6 to analyze the execution time and communication cost of the work-stealing algorithm. To conclude, in Section 7 we briefly discuss how the theoretical ideas in this paper have been applied to the Cilk programming language and runtime system [8, 25], as well as make some concluding remarks.

2 A model of multithreaded computation

This section reprises the graph-theoretic model of multithreaded computation introduced in [10]. 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 [13] and Graham [27, 28] on dag scheduling.

A multithreaded computation is composed of a set of threads, each of which is a sequential ordering of unit-time instructions. The instructions are connected by ***dependency*** edges, which provide a partial ordering on which instructions must execute before which other instructions. In Figure 1, for example, each shaded block is a thread with circles representing instructions and the horizontal edges, called ***continue*** edges, representing the sequential ordering. Thread Γ_5 of this example contains 3 instructions: v_{10}, v_{11}, and v_{12}. The instructions of a thread must execute in this sequential order from the first (leftmost) instruction to the last (rightmost) instruction. In order to execute a thread, we allocate for it a chunk of memory, called an ***activation frame***, that the instructions of the thread can use to store the values on which they compute.

A P-processor ***execution schedule*** for a multithreaded computation determines which processors of a P-processor parallel computer execute which instructions at each step. An execution schedule depends on the particular multithreaded computation and the number P of processors. In any given step of an execution schedule, each processor executes at most one instruction.

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

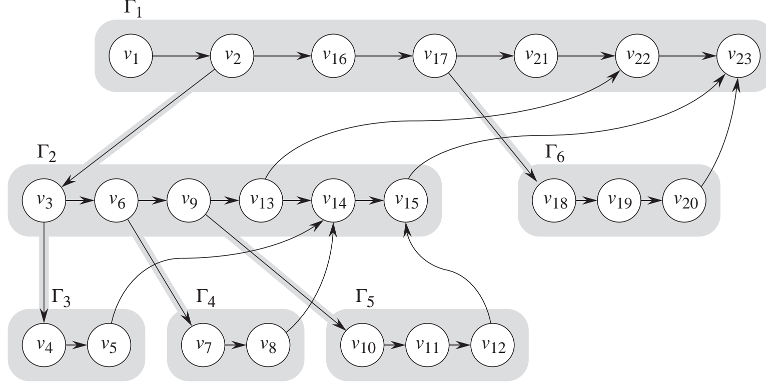


Figure 1: A multithreaded computation. This computation contains 23 instructions $v_1, v_2, \ldots, v_{23}$ and 6 threads $\Gamma_1, \Gamma_2, \ldots, \Gamma_6$.

threads are organized into a ***spawn tree*** as indicated in Figure 1 by the downward-pointing, shaded dependency edges, called ***spawn*** edges, that connect threads to their spawned children. The spawn tree is the parallel analog of a call tree. In our example computation, the spawn tree's ***root*** thread Γ_1 has two children, Γ_2 and Γ_6, and thread Γ_2 has three children, Γ_3, Γ_4, and Γ_5. Threads Γ_3, Γ_4, Γ_5, and Γ_6, which have no children, are ***leaf*** threads.

Each spawn edge goes from a specific instruction—the instruction that actually does the spawn operation—in the parent thread to the first instruction of the child thread. An execution schedule must obey this edge in that no processor may execute an instruction in a spawned child thread until after the spawning instruction in the parent thread has been executed. In our example computation (Figure 1), due to the spawn edge (v_6, v_7), instruction v_7 cannot be executed until after the spawning instruction v_6. Consistent with our unit-time model of instructions, a single instruction may spawn at most one child. When the spawning instruction executes, it allocates an activation frame for the new child thread. Once a thread has been spawned and its frame has been allocated, we say the thread is ***alive*** or ***living***. When the last instruction of a thread executes, it deallocates its frame and the thread ***dies***.

An execution schedule generally respects other dependencies besides those represented by continue and spawn edges. Consider an instruction that produces a data value to be consumed by another instruction. Such a producer/consumer relationship precludes the consuming instruction from executing until after the producing instruction. To enforce such orderings, other dependency edges, called ***join*** edges, may be required, as shown in Figure 1 by the curved edges. If the execution of a thread arrives at a consuming instruction before the producing instruction has executed, execution of the consuming thread cannot continue—the thread ***stalls***. Once the producing instruction executes, the join dependency is ***resolved***, which ***enables*** the consuming thread to resume its execution—the thread becomes ***ready***. A multithreaded computation does not model the means by which join dependencies get resolved or by which unresolved join dependencies get detected. In implementation, resolution and detection can be accomplished using mechanisms such as join counters [8], futures [30], or I-structures [2].

We make two technical assumptions regarding join edges. We first assume that each instruction has at most a constant number of join edges incident on it. This assumption

is consistent with our unit-time model of instructions. The second assumption is that no join edges enter the instruction immediately following a spawn. This assumption means that when a parent thread spawns a child thread, the parent cannot immediately stall. It continues to be ready to execute for at least one more instruction.

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

We make the simplifying assumption that a parent thread remains alive until all its children die, and thus, a thread does not deallocate its activation frame until all its children's frames have been deallocated. Although this assumption is not absolutely necessary, it gives the execution a natural structure, and it will simplify our analyses of space utilization. In accounting for 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 (or if such storage is available, then we do not account for it). Therefore, the space used at a given time in executing a computation is the total size of all frames used by all living threads at that time, and the total space used in executing a computation is the maximum such value over the course of the execution.

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

A given multithreaded program when run on a given input can sometimes generate more than one multithreaded computation. In that case, we say the program is ***nondeterministic***. If the same multithreaded computation is generated by the program on the input no matter how the computation is scheduled, then the program is ***deterministic***. In this paper, we shall analyze multithreaded computations, not multithreaded programs. Specifically, we shall not worry about how the multithreaded computation is generated. Instead, we shall study its properties in an *a posteriori* fashion.

Because multithreaded computations with arbitrary dependencies can be impossible to schedule efficiently [10], we study subclasses of general multithreaded computations in which the kinds of syncrhonizations that can occur are restricted. A ***strict*** multithreaded computation is one in which all join edges from a thread go to an ancestor of the thread in the activation tree. In a strict computation, the only edge into a subtree (emanating from outside the subtree) is the spawn edge that spawns the subtree's root thread. For example, the computation of Figure 1 is strict, and the only edge into the subtree rooted at Γ_2 is the spawn edge (v_2, v_3). Thus, strictness means that a thread 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 join 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 join edges from a subtree (of the spawn tree) emanate from the subtree's root. The example compu-

tation of Figure 1 is fully strict. 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 [5].

We quantify and bound the execution time of a computation on a P-processor parallel computer in terms of the computation's "work" and "critical-path length." We define the ***work*** of the computation to be the total number of instructions and the ***critical-path length*** to be the length of a longest directed path in the dag. Our example computation (Figure 1) has work 23 and critical-path length 10. For a given computation, let $T(\mathcal{X})$ denote the time to execute the computation using P-processor execution schedule $\mathcal{X}$, and let

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

denote the minimum execution time with P processors—the minimum being taken over all P-processor execution schedules for the computation. Then T_1 is the work of the computation, since a 1-processor computer can only execute one instruction at each step, and T_∞ is the critical-path length, since even with arbitrarily many processors, each instruction on a path must execute serially. Notice that we must have $T_P \geq T_1/P$, because P processors can execute only P instructions per time step, and of course, we must have $T_P \geq T_\infty$.

Early work on dag scheduling by Brent [13] and Graham [27, 28] shows that there exist P-processor execution schedules $\mathcal{X}$ with $T(\mathcal{X}) \leq T_1/P + T_\infty$. As the sum of two lower bounds, this upper bound is universally optimal to within a factor of 2. The following theorem, proved in [10, 20], 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 instructions are ready, then P instructions execute, and if fewer than P instructions are ready, then all execute.

Theorem 1 (The greedy-scheduling theorem) *For any multithreaded computation with work T_1 and critical-path length T_∞, and for any number P of processors, any greedy P-processor execution schedule $\mathcal{X}$ achieves $T(\mathcal{X}) \leq T_1/P + T_\infty$.* ■

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

To quantify the space used by a given execution schedule of a computation, we define the ***stack depth*** of a thread to be the sum of the sizes of the activation frames of all its ancestors, including itself. The ***stack depth*** of a multithreaded computation is the maximum stack depth of any of its threads. We shall denote by S_1 the minimum amount of space possible for any 1-processor execution of a multithreaded computation, which is equal to the stack depth of the computation. Let $S(\mathcal{X})$ denote the space used by a P-processor execution schedule $\mathcal{X}$ of a multithreaded computation. We shall be interested in those execution schedules that exhibit at most ***linear expansion of space***, that is, $S(\mathcal{X}) = O(S_1 P)$, which is existentially optimal to within a constant factor [10].

3 The busy-leaves property

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

In this section, we show that for any number P of processors and any strict multithreaded computation with work T_1, critical-path length T_∞, and stack depth S_1, there exists a P-processor execution schedule $\mathcal{X}$ that achieves time $T(\mathcal{X}) \leq T_1/P + T_\infty$ and space $S(\mathcal{X}) \leq S_1 P$ simultaneously. We give a simple online P-processor parallel algorithm—the ***Busy-Leaves Algorithm***—to compute such a schedule. This simple algorithm will form the basis for the randomized work-stealing algorithm presented in Section 4.

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

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

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

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

			processor activity			
step	thread pool		p_1		p_2	
1			$\mathbf{\Gamma}_1$:	v_1		
2				v_2		
3			$\mathbf{\Gamma}_2$:	v_3	$\mathbf{\Gamma}_1$:	v_{16}
4		$\mathbf{\Gamma}_2$	$\mathbf{\Gamma}_3$:	v_4		v_{17}
5	$\mathbf{\Gamma}_1$	$\mathbf{\Gamma}_2$		v_5	$\mathbf{\Gamma}_6$:	v_{18}
6	$\mathbf{\Gamma}_1$		$\mathbf{\Gamma}_2$:	v_6		v_{19}
7	$\mathbf{\Gamma}_1$	$\mathbf{\Gamma}_2$	$\mathbf{\Gamma}_4$:	v_7		v_{20}
8		$\mathbf{\Gamma}_2$		v_8	$\mathbf{\Gamma}_1$:	v_{21}
9	Γ_1		$\mathbf{\Gamma}_2$:	v_9		
10	Γ_1		$\mathbf{\Gamma}_5$:	v_{10}	$\mathbf{\Gamma}_2$:	v_{13}
11	$\mathbf{\Gamma}_1$			v_{11}		v_{14}
12		Γ_2		v_{12}	$\mathbf{\Gamma}_1$:	v_{22}
13	Γ_1		$\mathbf{\Gamma}_2$:	v_{15}		
14			$\mathbf{\Gamma}_1$:	v_{23}		

Figure 2: A 2-processor execution schedule computed by the Busy-Leaves Algorithm for the computation of Figure 1. This schedule lists the living threads in the global thread pool at each step just after each idle processor has removed a ready thread. It also lists the ready thread being worked on and the instruction executed by each of the 2 processors, p_1 and p_2, at each step. Living threads that are ready are listed in bold. The other living threads are stalled.

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

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

Figure 2 illustrates these three rules in a 2-processor execution schedule computed by the Busy-Leaves Algorithm on the computation of Figure 1. Rule ❶: At step 2, processor p_1 working on thread Γ_1 executes v_2 which spawns the child Γ_2, so p_1 places Γ_1 back in the pool (to be picked up at the beginning of the next step by the idle p_2) and begins the next step working on Γ_2. Rule ❷: At step 8, processor p_2 working on thread Γ_1 executes v_{21} and Γ_1 stalls, so p_2 returns Γ_1 to the pool and begins the next step idle (and remains idle since the thread pool contains no ready threads). Rule ❸: At step 13, processor p_1 working on Γ_2 executes v_{15} and Γ_2 dies, so p_1 retrieves the parent Γ_1 from the pool and begins the next step working on Γ_1.

Besides being greedy, for any strict computation, the schedule computed by the Busy-Leaves Algorithm maintains the ***busy-leaves property***: at every time step during the execution, every leaf in the "spawn subtree" has a processor working on it. We define the ***spawn subtree*** at any time step t to be the portion of the spawn tree consisting of just

those threads that are alive at step t. To restate the busy-leaves property, at every time step, every living thread that has no living descendants has a processor working on it. We shall now prove this fact and show that it implies linear expansion of space. It is worth noting that not every multithreaded computation has a schedule that maintains the busy-leaves property, but every strict multithreaded computation does. We begin by showing that any schedule that maintains the busy-leaves property exhibits linear expansion of space.

Lemma 2 *For any multithreaded computation with stack depth S_1, any P-processor execution schedule $\mathcal{X}$ that maintains the busy-leaves property uses space bounded by $S(\mathcal{X}) \leq S_1 P$.*

Proof: The busy-leaves property implies that at all time steps t, the spawn subtree has at most P leaves. For each such leaf, the space used by it and all of its ancestors is at most S_1, and therefore, the space in use at any time step t is at most $S_1 P$.

For schedules that maintain the busy-leaves property, the upper bound $S_1 P$ is conservative. By charging S_1 space for each busy leaf, we may be overcharging. For some computations, by knowing that the schedule preserves the busy-leaves property, we can appeal directly to the fact that the spawn subtree never has more than P leaves to obtain tight bounds on space usage [6].

We finish this section by showing that for strict computations, the Busy-Leaves Algorithm computes a schedule that is both greedy and maintains the busy-leaves property.

Theorem 3 *For any number P of processors and any strict multithreaded computation with work T_1, critical-path length T_∞, and stack depth S_1, the Busy-Leaves Algorithm computes a P-processor execution schedule $\mathcal{X}$ whose execution time satisfies $T(\mathcal{X}) \leq T_1/P + T_\infty$ and whose space satisfies $S(\mathcal{X}) \leq S_1 P$.*

Proof: The time bound follows directly from the greedy-scheduling theorem (Theorem 1), since the Busy-Leaves Algorithm computes a greedy schedule. The space bound follows from Lemma 2 if we can show that the Busy-Leaves Algorithm maintains the busy-leaves property. We prove this fact by induction on the number of steps. At the first step of the algorithm, the spawn subtree contains just the root thread which is a leaf, and some processor is working on it. We must show that all of the algorithm rules preserve the busy-leaves property. When a processor has a thread Γ_a to work on, it executes instructions from that thread until it either spawns, stalls, or dies. Rule ❶: If Γ_a spawns a child Γ_b, then Γ_a is not a leaf (even if it was before) and Γ_b is a leaf. In this case, the processor works on Γ_b, so the new leaf is busy. Rule ❷: If Γ_a stalls, then Γ_a cannot be a leaf since in a strict computation, the unresolved dependency must come from a descendant. Rule ❸: If Γ_a dies, then its parent thread Γ_b may turn into a leaf. In this case, the processor works on Γ_b unless some other processor already is, so the new leaf is guaranteed to be busy.

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

4 A randomized work-stealing algorithm

In this section, we present an online, randomized work-stealing algorithm for scheduling multithreaded computations on a parallel computer. Also, we present an important structural lemma which is used at the end of this section to show that for fully strict computations, this algorithm causes at most a linear expansion of space. This lemma reappears in Section 6 to show that for fully strict computations, this algorithm achieves linear speedup and generates existentially optimal amounts of communication.

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

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

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

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

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

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

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

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

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

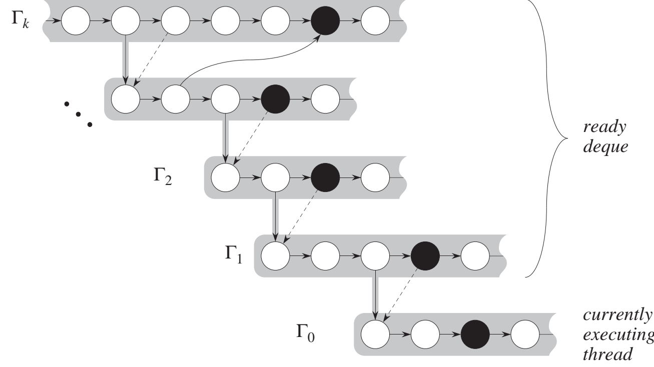


Figure 3: The structure of a processor's ready deque. The black instruction in each thread indicates the thread's currently ready instruction. Only thread Γ_k may have been worked on since it last spawned a child. The dashed edges are the "deque edges" introduced in Section 6.

We now state and prove an important lemma on the structure of threads in the ready deque of any processor during the execution of a fully strict computation. This lemma is used later in this section to analyze execution space and in Section 6 to analyze execution time and communication. Figure 3 illustrates the lemma.

Lemma 4 *In the execution of any fully strict multithreaded computation by the Work-Stealing Algorithm, consider any processor p and any given time step at which p is working on a thread. Let Γ_0 be the thread that p is working on, let k be the number of threads in p's ready deque, and let $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ denote the threads in p's ready deque ordered from bottom to top, so that Γ_1 is the bottommost and Γ_k is the topmost. If we have $k > 0$, then the threads in p's ready deque satisfy the following properties:*

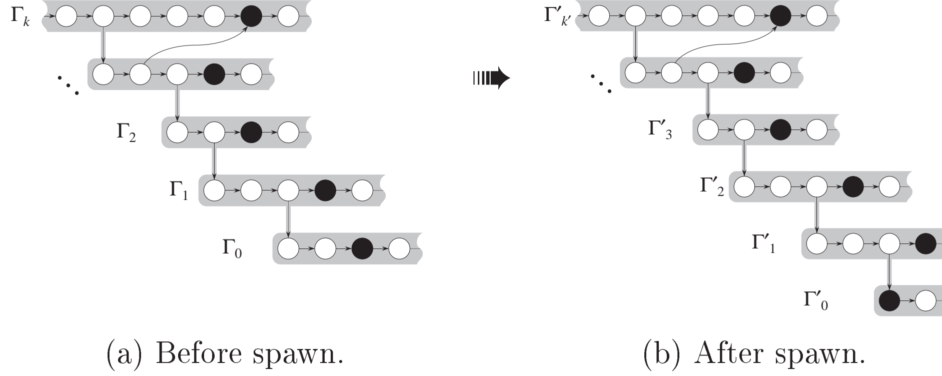
① *For $i = 1, 2, \ldots, k$, thread Γ_i is the parent of Γ_{i-1}.*

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

Proof: The proof is a straightforward induction on execution time. Execution begins with the root thread in some processor's ready deque and all other ready deques empty, so the lemma vacuously holds at the outset. Now, consider any step of the algorithm at which processor p executes an instruction from thread Γ_0. Let $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ denote the k threads in p's ready deque before the step, and suppose that either $k = 0$ or both properties hold. Let Γ'_0 denote the thread (if any) being worked on by p after the step, and let $\Gamma'_1, \Gamma'_2, \ldots, \Gamma'_{k'}$ denote the k' threads in p's ready deque after the step. We now look at the rules of the algorithm and show that they all preserve the lemma. That is, either $k' = 0$ or both properties hold after the step.

Rule ❶: If Γ_0 spawns a child, then p pushes Γ_0 onto the bottom of the ready deque and commences work on the child. Thus, Γ'_0 is the child, we have $k' = k + 1 > 0$, and for $j = 1, 2, \ldots, k'$, we have $\Gamma'_j = \Gamma_{j-1}$. See Figure 4. Now, we can check both properties. Property ①: If $k' > 1$, then for $j = 2, 3, \ldots, k'$, thread Γ'_j is the parent of Γ'_{j-1}, since before the spawn we have $k > 0$, which means that for $i = 1, 2, \ldots, k$, thread Γ_i is the parent of Γ_{i-1}.

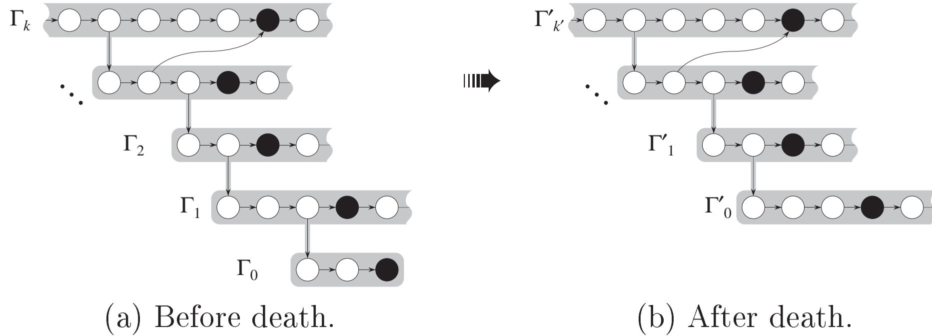
Moreover, Γ'_1 is obviously the parent of Γ'_0. Property ②: If $k' > 2$, then for $j = 2, 3, \ldots, k'-1$, thread Γ'_j has not been worked on since it spawned Γ'_{j-1}, because before the spawn we have $k > 1$, which means that for $i = 1, 2, \ldots, k-1$, thread Γ_i has not been worked on since it spawned Γ_{i-1}. Finally, thread Γ'_1 has not been worked on since it spawned Γ'_0, because the spawn only just occurred.



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

Figure 4: The ready deque of a processor before and after the thread Γ_0 that it is working on spawns a child. (Note that the threads Γ_0 and Γ'_0 are not actually in the deque; they are the threads being worked on before and after the spawn.)

Rules ❷ and ❸: If Γ_0 stalls or dies, then we have two cases to consider. If $k = 0$, the ready deque is empty, so the processor commences work stealing, and when the processor steals and begins work on a thread, we have $k' = 0$. If $k > 0$, the ready deque is not empty, so the processor pops the bottommost thread off the deque and commences work on it. Thus, we have $\Gamma'_0 = \Gamma_1$ (the popped thread) and $k' = k - 1$, and for $j = 1, 2, \ldots, k'$, we have $\Gamma'_j = \Gamma_{j+1}$. See Figure 5. Now, if $k' > 0$, we can check both properties. Property ①: For $j = 1, 2, \ldots, k'$, thread Γ'_j is the parent of Γ'_{j-1}, since for $i = 1, 2, \ldots, k$, thread Γ_i is the parent of Γ_{i-1}. Property ②: If $k' > 1$, then for $j = 1, 2, \ldots, k'-1$, thread Γ'_j has not been worked on since it spawned Γ'_{j-1}, because before the stall or death we have $k > 2$, which means that for $i = 2, 3, \ldots, k-1$, thread Γ_i has not been worked on since it spawned Γ_{i-1}.



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

Figure 5: The ready deque of a processor before and after the thread Γ_0 that it is working on dies. (Note that the threads Γ_0 and Γ'_0 are not actually in the deque; they are the threads being worked on before and after the death.)

Rule ❹: If Γ_0 enables a stalled thread, then due to the fully strict condition, that previously stalled thread must be Γ_0's parent. First, we observe that we must have $k = 0$. If

we have $k > 0$, then the processor's ready deque is not empty, and this parent thread must be bottommost in the ready deque. Thus, this parent thread is ready and Rule ❹ does not apply. With $k = 0$, the ready deque is empty and the processor places the parent thread on the bottom of the ready deque. We have $\Gamma'_0 = \Gamma_0$ and $k' = k + 1 = 1$ with Γ'_1 denoting the newly enabled parent. We only have to check the first property. Property ①: Thread Γ'_1 is obviously the parent of Γ'_0.

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

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

We conclude this section by bounding the space used by the Work-Stealing Algorithm executing a fully strict computation.

Theorem 5 *For any fully strict multithreaded computation with stack depth S_1, the Work-Stealing Algorithm run on a computer with P processors uses at most $S_1 P$ space.*

Proof: Like the Busy-Leaves Algorithm, the Work-Stealing Algorithm maintains the busy-leaves property: at every time step of the execution, every leaf in the current spawn subtree has a processor working on it. If we can establish this fact, then Lemma 2 completes the proof.

That the Work-Stealing Algorithm maintains the busy-leaves property is a simple consequence of Lemma 4. At every time step, every leaf in the current spawn subtree must be ready and therefore must either have a processor working on it or be in the ready deque of some processor. But Lemma 4 guarantees that no leaf thread sits in a processor's ready deque while the processor works on some other thread.

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

5 Atomic accesses and the recycling game

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

The ***atomic-access model*** is the machine model we use to analyze the Work-Stealing Algorithm. We assume that the machine is an asynchronous parallel computer with P

processors, and its memory can be either distributed or shared. Our analysis assumes that concurrent accesses to the same data structure are serially queued by an adversary, as in the atomic message-passing model of [36]. This assumption is more stringent than that in the model of Karp and Zhang [33]. They assume that if concurrent steal requests are made to a deque, in one time step, one request is satisfied and all the others are denied. In the atomic-access model, we also assume that one request is satisfied, but the others are queued by an adversary, rather than being denied. Moreover, from the collection of waiting requests for a given deque, the adversary gets to choose which is serviced and which continue to wait. The only constraint on the adversary is that if there is at least one request for a deque, then the adversary cannot choose that none be serviced.

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

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

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

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

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

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

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

the choices that the adversary makes about which balls to toss into bins and which to return to the reservoir, the total delay is unlikely to be large.

Lemma 6 *For any $\epsilon > 0$, with probability at least $1-\epsilon$, the total delay in the (P, M)-recycling game is $O(M + P \lg P + P \lg(1/\epsilon))$.*[1] *The expected total delay is at most M. In other words, the total delay incurred by M random requests made by P processors in the atomic-access model is $O(M + P \lg P + P \lg(1/\epsilon))$ 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 are queued in their bins in a first-in-first-out (FIFO) order. The adversary removes balls from the front of the queue, and when the adversary tosses a ball, it is placed on the back of the queue. If several balls are tossed into the same bin at the same step, they can be placed on the back of the queue in any order. The reason that assuming a FIFO discipline for queuing balls in a bin does not affect the total delay is that the number of balls in a given bin at a given step is the same no matter which ball is removed, and where balls are tossed has nothing to do with which ball is tossed.

For any given ball and any given step, the step either finishes with the the ball in a bin or in the reservoir. Define the ***delay*** of ball r to be the random variable 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

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

Thus, we have

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

[1] Greg Plaxton of the University of Texas, Austin has improved this bound to $O(M)$ for the case when $1/\epsilon$ is at most polynomial in M and P [40].

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

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

where $S' = S - \{(i,r)\}$, whence the claim (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, where e is the base of the natural logarithm, 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 $\Delta \ge \max\{2em,\ \lg P + \lg(1/\epsilon)\}$.

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

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

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 $\mathrm{E}[D] \leq M$.

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

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

Unlike in the Busy-Leaves Algorithm, the "ready pool" in the Work-Stealing Algorithm is distributed, and so there is no contention at a centralized data structure. Nevertheless, it is still possible for contention to arise when several thieves happen to descend on the same victim simultaneously. In this case, as we have indicated in the previous section, we make the conservative assumption that an adversary serially queues the work-stealing requests. We further assume that it takes unit time for a processor to respond to a work-stealing request. This assumption can be relaxed without materially affecting the results so that a work-stealing response takes any constant amount of time.

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

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

Lemma 7 *The execution of a fully strict multithreaded computation with work T_1 by the Work-Stealing Algorithm on a computer with P processors terminates with exactly T_1 dollars in the* Work *bucket.*

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

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

A ***round*** of 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 of 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. Moreover, since at most $P - 1$ steal attempts can be initiated in a single time step, each round contains fewer than $4P - 1$ steal attempts, and each round takes at least 4 steps.

The sequence of instructions that make up the delay sequence is defined with respect to an augmented dag obtained by modifying the original dag slightly. Let G denote the original dag, that is, the dag consisting of the computation's instructions as vertices and its continue, spawn, and join edges as edges. The augmented dag G' is the original dag G together with some new edges, as follows. For every set of instructions u, v, and w such that (u, v) is a spawn edge and (u, w) is a continue edge, the ***deque edge*** (w, v) is placed in G'. These deque edges are shown dashed in Figure 3. In Section 2 we made the technical assumption that instruction w has no incoming join edges, and so G' is a dag. If T_∞ is the length of a longest path in G, then the longest path in G' has length at most $2T_\infty$. It is worth pointing out that G' is only an analytical tool. The deque edges have no effect on the scheduling and execution of the computation by the Work-Stealing Algorithm.

The deque edges are the key to defining critical instructions. At any time step during the execution, we say that an unexecuted instruction v is ***critical*** if every instruction that precedes v (either directly or indirectly) in G' has been executed, that is, if for every instruction w such that there is a directed path from w to v in G', instruction w has been

executed. A critical instruction must be ready, since G' contains every edge of G, but a ready instruction may or may not be critical. Intuitively, the structural properties of a ready deque enumerated in Lemma 4 guarantee that if a thread is deep in a ready deque, then its current instruction cannot be critical, because the predecessor of the thread's current instruction across the deque edge has not yet been executed.

We now formalize our definition of a delay sequence.

Definition 8 *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 G'. Specifically, for $i = 1, 2, \ldots, L-1$, the edge (u_i, u_{i+1}) belongs to G', instruction u_1 has no incoming edges in G' (instruction u_1 must be the first instruction of the root thread), and instruction u_L has no outgoing edges in G' (instruction u_L must be the last instruction of the root thread).*

- *R is a positive integer number of steal-attempt rounds.*

- *$\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \ldots, \pi_L, \pi'_L)$ is a partition of R (that is $R = \sum_{i=1}^{L}(\pi_i + \pi'_i)$), such that $\pi'_i \in \{0, 1\}$ for each $i = 1, 2, \ldots, L$.*

The partition Π induces a partition of a sequence of R rounds as follows. The first piece of the partition corresponds to the first π_1 rounds. The second piece corresponds to the next π'_1 consecutive rounds after the first π_1 rounds. The third piece corresponds to the next π_2 consecutive rounds after the first $(\pi_1 + \pi'_1)$ rounds, and so on. We are interested primarily in the pieces corresponding to the π_i, not the π'_i, and so we define the ith ***group*** of rounds to be the π_i consecutive rounds starting after the r_ith round, where $r_i = \sum_{j=1}^{i-1}(\pi_j + \pi'_j)$. Because Π is a partition of R and $\pi'_i \in \{0, 1\}$, for $i = 1, 2, \ldots, L$, we have

$$\sum_{i=1}^{L} \pi_i \geq R - L \ . \tag{5}$$

We say that a given round of steal attempts ***occurs*** while instruction v is critical if all of the steal attempts that comprise the round are initiated at time steps when v is critical. In other words, v must be critical throughout the entire round. A delay sequence (U, R, Π) is said to ***occur*** during an execution if for each $i = 1, 2, \ldots, L$, all π_i rounds in the ith group occur while instruction u_i is critical. In other words, u_i must be critical throughout all π_i rounds.

The following lemma states that if at least R rounds take place during an execution, then some delay sequence (U, R, Π) must occur. In particular, if we look at any execution in which at least R rounds occur, then we can identify a path $U = (u_1, u_2, \ldots, u_L)$ in the dag G' and a partition $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \ldots, \pi_L, \pi'_L)$ of the first R rounds, such that for each $i = 1, 2, \ldots L$, all of the π_i rounds in the ith group occur while u_i is critical. Each π'_i indicates whether u_i is critical at the beginning of a round but gets executed before the round ends. Such a round cannot be part of any group, because no instruction is critical throughout.

Lemma 9 *Consider the execution of a fully strict multithreaded computation with critical-path length T_∞ by the Work-Stealing Algorithm on a computer with P processors. If at least $4PR$ steal attempts occur during the execution, then some delay sequence (U, R, Π) must occur.*

Proof: For a given execution in which at least $4PR$ steal attempts take place, we construct a delay sequence (U, R, Π) and show that it occurs. With at least $4PR$ steal attempts, there must be at least R rounds. We construct the delay sequence by first identifying a set of instructions on a directed path in G' such that for every time step during the execution, one of these instructions is critical. Then, we partition the first R rounds according to when each round occurs relative to when each instruction on the path is critical.

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

Now, to construct the partition $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \ldots, \pi_L, \pi'_L)$, we partition the sequence of the first R rounds according to when each round occurs. We would like our partition to be such that for each round (among the first R rounds), we have the property that if the round occurs while some instruction u_i is critical, then the round belongs to the ith group. Start with π_1, and let π_1 equal the number of rounds that occur while u_1 is critical. All of these rounds are consecutive at the beginning of the sequence, so these rounds comprise the 1st group—that is, they are the π_1 consecutive rounds starting after the $r_1 = 0$ first rounds. Next, if the round that immediately follows those first π_1 rounds begins after u_1 has been executed, then we set $\pi'_1 = 0$, and we go on to π_2. Otherwise, that round begins while u_1 is critical and ends after u_1 is executed (for otherwise, it would be part of the first group), so we set $\pi'_1 = 1$, and we go on to π_2. For π_2, we let π_2 equal the number of rounds that occur while u_2 is critical. Note that all of these rounds are consecutive beginning after the first $r_2 = \pi_1 + \pi'_1$ rounds, so these rounds comprise the 2nd group. We continue in this fashion, letting each π_i be the number of rounds that occur while u_i is critical and letting each π'_i be the number of rounds that begin while u_i is critical but do not end until after u_i is executed. As an example, we may have a round that begins while u_i is critical and then ends while u_{i+2} is critical, and in this case, we set $\pi'_i = 1$ and $\pi'_{i+1} = 0$. In this example, the $(i + 1)$st group is empty, so we also set $\pi_{i+1} = 0$.

We conclude the proof by verifying that the (U, R, Π) as just constructed is a delay sequence and that it occurs. By construction, U is a maximal path in G'. Now considering Π, we observe that each round among the first R rounds is counted exactly once in either a π_i or a π'_i, so Π is indeed a partition of R. Moreover, for $i = 1, 2, \ldots, L$, at most one round can begin while the instruction u_i is critical and end after u_i is executed, so we have $\pi'_i \in \{0, 1\}$. Thus, (U, R, Π) is a delay sequence. Finally, we observe that, by construction, for $i = 1, 2, \ldots, L$, the π_i rounds that comprise the ith group all occur while the instruction u_i is critical. Therefore, the delay sequence (U, R, Π) occurs.

We now establish that a critical instruction is unlikely to remain critical across a modest number of rounds. Specifically, we first show that a critical instruction must be the ready

instruction of a thread that is near the top of its processor's ready deque. We then use this fact to show that after $O(1)$ rounds, a critical instruction is very likely to be executed.

Lemma 10 *At every time step during the execution of a fully strict multithreaded computation by the Work-Stealing Algorithm, each critical instruction is the ready instruction of a thread that has at most 1 thread above it in its processor's ready deque.*

Proof: Consider any time step, and let u_0 be the critical instruction of a thread Γ_0. Since u_0 is critical, Γ_0 is ready. Hence, for some processor p, either Γ_0 is in p's ready deque or Γ_0 is being worked on by p. If Γ_0 has more than 1 thread above it in p's ready deque, then Lemma 4 guarantees that each of the at least 2 threads above Γ_0 in p's ready deque is an ancestor of Γ_0. Let $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ denote Γ_0's ancestor threads, where Γ_1 is the parent of Γ_0 and Γ_k is the root thread. Further, for $i = 1, 2, \ldots, k$, let u_i denote the instruction of thread Γ_i that spawned thread Γ_{i-1}, and let w_i denote u_i's successor instruction in thread Γ_i. Because of the deque edges, each instruction w_i is a predecessor of u_0 in G', and consequently, since u_0 is critical, each instruction w_i has been executed. Moreover, because each w_i is the successor of the spawn instruction u_i in thread Γ_i, each thread Γ_i for $i = 1, 2, \ldots, k$ has been worked on since the time step at which it spawned thread Γ_{i-1}. But Lemma 4 guarantees that only the topmost thread in p's ready deque can have this property. Thus, Γ_1 is the only thread that can possibly be above Γ_0 in p's ready deque.

Lemma 11 *Consider the execution of any fully strict multithreaded computation by the Work-Stealing Algorithm on a parallel computer with $P \geq 2$ processors. For any instruction v and any number $r \geq 2$ of steal-attempt rounds, the probability that any particular set of r rounds occur while the instruction v is critical is at most the probability that only* 0 *or* 1 *of the steal attempts initiated in the first $r-1$ of these rounds choose v's processor, which is at most e^{-2r+3}.*

Proof: Let t_a denote the first time step at which instruction v is critical, and let p denote the processor in whose ready deque v's thread resides at time step t_a. Consider any particular set of r rounds, and suppose that they all occur while instruction v is critical. Now, consider the steal attempts that comprise the first $r-1$ of these rounds, of which there must be at least $3P(r-1)$. Let t_b denote the time step just after the time step at which the last of these steal attempts is initiated. Note that because the last round, like every round, must take at least two (in fact, four) steps, the time step t_b must occur before the time step at which instruction v is executed.

We shall first show that of these $3P(r-1)$ steal attempts initiated while instruction v is critical and at least 2 time steps before v is executed, at most 1 of them can choose processor p as its target, for otherwise, v would be executed at or before t_b. Recall from Lemma 10 that instruction v is the ready instruction of a thread Γ, which has at most 1 thread above it in p's ready deque as long as v is critical.

If Γ has no threads above it, then another thread cannot be placed above it until after instruction v is executed, since only by processor p executing instructions from Γ can another thread be placed above it in its ready deque. Consequently, if a steal attempt targeting processor p is initiated at some time step $t \geq t_a$, we are guaranteed that instruction v is

executed at a time step no later than t, either by thread Γ being stolen and executed or by p executing the thread itself.

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

Thus, we must have $3P(r-1)$ 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 $3P(r-1)$ steal attempts chooses processor p is

$$\begin{aligned}
&\left(1 - \frac{1}{P}\right)^{3P(r-1)} + 3P(r-1)\left(\frac{1}{P}\right)\left(1 - \frac{1}{P}\right)^{3P(r-1)-1} \\
&= \left(1 + 3(r-1)\frac{P}{P-1}\right)\left(1 - \frac{1}{P}\right)^{3P(r-1)} \\
&\leq (6r-5)\left(1 - \frac{1}{P}\right)^{3P(r-1)} \\
&\leq (6r-5)\, e^{-3(r-1)} \\
&\leq e^{-2r+3}
\end{aligned}$$

for $r \geq 2$.

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

Lemma 12 *Consider the execution of any fully strict multithreaded computation with critical-path length T_∞ by the Work-Stealing Algorithm 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 9, we know that if at least $4PR$ steal attempts occur, then some delay sequence (U, R, Π) must occur. Consider a particular delay sequence (U, R, Π) having $U = (u_1, u_2, \ldots, u_L)$ and $\Pi = (\pi_1, \pi'_1, \pi_2, \pi'_2, \ldots, \pi_L, \pi'_L)$, with $L \leq 2T_\infty$. We shall compute the probability that (U, R, Π) occurs.

Such a sequence occurs if for $i = 1, 2, \ldots, L$, each instruction u_i is critical throughout all π_i rounds in the ith group. From Lemma 11, we know that the probability of the π_i rounds in the ith group all occurring while a given instruction u_i is critical is at most the probability that only 0 or 1 of the steal attempts initiated in the first $\pi_i - 1$ of these rounds

choose v's processor, which is at most $e^{-2\pi_i+3}$, provided $\pi_i \geq 2$. (For those values of i with $\pi_i < 2$, we shall use 1 as an upper bound on this probability.) Moreover, since the targets of the work-steal attempts in the π_i rounds of the ith group 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:

$$
\begin{aligned}
&\Pr\{(U, R, \Pi) \text{ occurs}\} \\
&\quad = \prod_{1 \leq i \leq L} \Pr\{\text{the } \pi_i \text{ rounds of the } i\text{th group occur while } u_i \text{ is critical}\} \\
&\quad \leq \prod_{\substack{1 \leq i \leq L \\ \pi_i \geq 2}} e^{-2\pi_i+3} \\
&\quad \leq \exp\left[-2\left(\sum_{\substack{1 \leq i \leq L \\ \pi_i \geq 2}} \pi_i\right) + 3L\right] \\
&\quad = \exp\left[-2\left(\sum_{1 \leq i \leq L} \pi_i - \sum_{\substack{1 \leq i \leq L \\ \pi_i < 2}} \pi_i\right) + 3L\right] \\
&\quad \leq e^{-2((R-L)-L)+3L} \\
&\quad = e^{-2R+7L} ,
\end{aligned}
$$

where the second-to-last line follows from Inequality (5).

To bound the probability of some delay sequence (U, R, Π) occurring, we need to count the number of such delay sequences and multiply by the probability that a particular such sequence occurs. The directed path U in the modified dag G' starts at the first instruction of the root thread and ends at the last instruction of the root thread. If the original dag has degree d, then G' has degree at most $d + 1$. Consistent with our unit-time assumption for instructions, we assume that the degree d is a constant. Since the length of a longest path in G' is at most $2T_\infty$, there are at most $(d+1)^{2T_\infty}$ ways of choosing the path $U = (u_1, u_2, \ldots, u_L)$. There are at most $\binom{2L+R}{R} \leq \binom{4T_\infty+R}{R}$ ways to choose Π, since Π partitions R into $2L$ pieces. As we have just shown, a given delay sequence has at most an $e^{-2R+7L} \leq e^{-2R+14T_\infty}$ chance of occurring. Multiplying these three factors together bounds the probability that any delay sequence (U, R, Π) occurs by

$$(d+1)^{2T_\infty} \binom{4T_\infty + R}{R} e^{-2R+14T_\infty} , \tag{6}$$

which is at most ϵ for $R = cT_\infty \lg d + \lg(1/\epsilon)$, where c is a sufficiently large constant. Thus, the probability that at least $4PR = \Theta(P(T_\infty \lg d + \lg(1/\epsilon))) = \Theta(P(T_\infty + \lg(1/\epsilon)))$ 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 the Work-Stealing Algorithm, and we complete the proof by bounding the number of dollars in the Wait bucket.

Theorem 13 *Consider the execution of any fully strict multithreaded computation with work T_1 and critical-path length T_∞ by the Work-Stealing Algorithm on a parallel computer with P processors. The expected running time, including scheduling overhead, is $T_1/P + O(T_\infty)$. Moreover, for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the execution time on P processors is $T_1/P + O(T_\infty + \lg P + \lg(1/\epsilon))$.*[2]

Proof: Lemmas 7 and 12 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 6 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 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 the Work-Stealing Algorithm performs in a distributed model. The analysis makes the assumption that at most a constant number of bytes need be communicated along a join edge to resolve the dependency.

Theorem 14 *Consider the execution of any fully strict multithreaded computation with critical-path length T_∞ by the Work-Stealing Algorithm on a parallel computer with P processors. Then, the total number of bytes communicated has expectation $O(PT_\infty(1+n_d)S_{max})$ where n_d is the maximum number of join edges from a thread to its parent and S_{max} is the size in bytes of the largest activation frame in the computation. Moreover, for any $\epsilon > 0$, the probability is at least $1 - \epsilon$ that the total communication incurred is $O(P(T_\infty + \lg(1/\epsilon))(1 + n_d)S_{max})$.*

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

The communication bounds in this theorem are existentially tight, in that there exist fully strict computations that require $\Omega(PT_\infty(1 + n_d)S_{\max})$ total communication for any execution schedule that achieves linear speedup. This result follows directly from a theorem of Wu and Kung [47], who showed that divide-and-conquer computations—a special case of fully strict computations with $n_d = 1$—require this much communication.

[2]With Plaxton's bound [40] for Lemma 6, this bound becomes $T_1/P + O(T_\infty)$, whenever $1/\epsilon$ is at most polynomial in M and P.

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

7 Conclusion

How practical are the methods analyzed in this paper? We have been actively engaged in building a C-based language called ***Cilk*** (pronounced "silk") for programming multithreaded computations [5, 8, 25, 32, 42]. Cilk is derived from the PCM "Parallel Continuation Machine" system [29], which was itself partly inspired by the research reported here. The Cilk runtime system employs the Work-Stealing Algorithm described in this paper. Because Cilk employs a provably efficient scheduling algorithm, Cilk delivers guaranteed performance to user applications. Specifically, we have found empirically that the performance of an application written in the Cilk language can be predicted accurately using the model $T_1/P + T_\infty$.

The Cilk system currently runs on contemporary shared-memory multiprocessors, such as the Sun Enterprise, the Silicon Graphics Origin, the Intel Quad Pentium, and the DEC Alphaserver. (Earlier versions of Cilk ran on the Thinking Machines CM-5 MPP, the Intel Paragon MPP, and the IBM SP-2.) To date, applications written in Cilk include protein folding [38], graphic rendering [45], backtrack search, and the `⋆Socrates` chess program [31], which won second prize in the 1995 ICCA World Computer Chess Championship running on a 1824-node Paragon at Sandia National Laboratories. Our more recent chess program, Cilkchess, won the 1996 Dutch Open Computer Chess Tournament. A team programming in Cilk won First Prize (undefeated) in the ICFP'98 Programming Contest sponsored by the International Conference on Functional Programming.

As part of our research, we have implemented a prototype runtime system for Cilk on networks of workstations. This runtime system, called *Cilk-NOW* [5, 11, 35], supports adaptive parallelism, where processors in a workstation environment can join a user's computation if they would be otherwise idle and yet be available immediately to leave the computation when needed again by their owners. Cilk-NOW also supports transparent fault tolerance, meaning that the user's computation can proceed even in the face of processors crashing, and yet the programmer writes the code in a completely fault-oblivious fashion. A more recent distributed implementation for clusters of SMP's is described in [42].

We have also investigated other topics related to Cilk, including distributed shared memory [6, 7, 24, 26] and debugging tools [17, 18, 22, 45]. Up-to-date information, papers, and software releases can be found on the World Wide Web at `http://supertech.lcs.mit.edu/cilk`.

For the case of shared-memory multiprocessors, we have recently generalized the time bound (but not the space or communication bounds) along two dimensions [1]. First, we have shown that for arbitrary (not necessarily fully strict or even strict) multithreaded computations, the expected execution time is $O(T_1/P + T_\infty)$. This bound is based on a new structural lemma and an amortized analysis using a potential function. Second, we have developed a nonblocking implementation of the work-stealing algorithm, and we have analyzed its execution time for a multiprogrammed environment in which the computation

executes on a set of processors that grows and shrinks over time. This growing and shrinking is controlled by an adversary. In case the adversary chooses not to grow or shrink the set of processors, the bound specializes to match our previous bound. The nonblocking work stealer has been implemented in the ***Hood*** user-level threads library [12, 39]. Up-to-date information, papers, and software releases can be found on the World Wide Web at `http://www.cs.utexas.edu/users/hood`.

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 the Work-Stealing Algorithm, which improved our previous bounds. Thanks to Greg Plaxton of University of Texas, Austin for technical comments on our probabilistic analyses. Thanks to the anonymous referees, 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] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, Puerto Vallarta, Mexico, June 1998.

[2] 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.

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

[4] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 12–23, Newport, Rhode Island, June 1997.

[5] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.

[6] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[7] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the Tenth International Parallel Processing Symposium (IPPS)*, pages 132–141, Honolulu, Hawaii, April 1996.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

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

[10] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, February 1998.

[11] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, January 1997.

[12] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. Technical Report TR-98-13, The University of Texas at Austin, Department of Computer Sciences, May 1998.

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

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

[15] F. Warren Burton. Guaranteeing good memory bounds for parallel programs. *IEEE Transactions on Software Engineering*, 22(10), October 1996.

[16] 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.

[17] Guang-Ien Cheng. Algorithms for data-race detection in multithreaded programs. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[18] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Tenth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998.

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

[20] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[21] R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system. *Advances in Computer Chess 7*, pages 203–219, 1993.

[22] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

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

[24] Matteo Frigo. The weakest reasonable memory model. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1998.

[25] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

[26] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 240–249, Puerto Vallarta, Mexico, June 1998.

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

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

[29] 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.

[30] 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.

[31] Chris Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge*, Rutgers University, New Jersey, October 1994.

[32] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[33] 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.

[34] Bradley C. Kuszmaul. *Synchronized MIMD Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-645.

[35] Philip Lisiecki. Macroscheduling in the Cilk network of workstations environment. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1996.

[36] 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 (SPAA)*, pages 154–163, Velen, Germany, June 1993.

[37] 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.

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

[39] Dionysios P. Papadopoulos. Hood: A user-level thread library for multiprogramming multiprocessors. Master's thesis, Department of Computer Sciences, The University of Texas at Austin, August 1998.

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

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

[42] Keith H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[43] 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 (SPAA)*, pages 237–245, Hilton Head, South Carolina, July 1991.

[44] 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.

[45] Andrew F. Stark. Debugging multithreaded programs that incorporate user-level locking. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

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

[47] 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 (FOCS)*, pages 151–162, San Juan, Puerto Rico, October 1991.

[48] 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.