Thread Scheduling for Multiprogrammed Multiprocessors

Nimar S. Arora Robert D. Blumofe C. Greg Plaxton
Department of Computer Science, University of Texas at Austin
{nimar,rdb,plaxton}@cs.utexas.edu

Abstract

We present a user-level thread scheduler for shared-memory multiprocessors, and we analyze its performance under multiprogramming. We model multiprogramming with two scheduling levels: our scheduler runs at user-level and schedules threads onto a fixed collection of processes, while below, the operating system kernel schedules processes onto a fixed collection of processors. We consider the kernel to be an adversary, and our goal is to schedule threads onto processes such that we make efficient use of whatever processor resources are provided by the kernel.

Our thread scheduler is a non-blocking implementation of the work-stealing algorithm. For any multithreaded computation with work T_1 and critical-path length T_∞, and for any number P of processes, our scheduler executes the computation in expected time $O(T_1/P_A + T_\infty P/P_A)$, where P_A is the average number of processors allocated to the computation by the kernel. This time bound is optimal to within a constant factor, and achieves linear speedup whenever P is small relative to the parallelism T_1/T_∞.

1 Introduction

Operating systems for shared-memory multiprocessors support multiprogrammed workloads in which a mix of serial and parallel applications may execute concurrently. For example, on a multiprocessor workstation, a parallel design verifier may execute concurrently with other serial and parallel applications, such as the design tool's user interface, compilers, editors, and web clients. For parallel applications, operating systems provide system calls for the creation and synchronization of multiple threads, and they provide high-level multithreaded programming support with parallelizing compilers and threads libraries. In addition, programming languages, such as Cilk [7, 21] and Java [3], support multithreading with linguistic abstractions. A major factor in the performance of such multithreaded parallel applications is the operation of the thread scheduler.

Prior work on thread scheduling [4, 5, 8, 13, 14] has dealt exclusively with non-multiprogrammed environments in which a multithreaded computation executes on P dedicated processors. Such scheduling algorithms dynamically map threads onto the processors with the goal of achieving P-fold speedup. Though such algorithms will work in some multiprogrammed environments, in particular those that employ static space partitioning [15, 30] or coscheduling [18, 30, 33], they do not work in the multiprogrammed environments being supported by modern shared-memory multiprocessors and operating systems [9, 15, 17, 23]. The problem lies in the assumption that a fixed collection of processors are fully available to perform a given computation.

This research is supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0150 from the U.S. Air Force Research Laboratory. In addition, Greg Plaxton is supported by the National Science Foundation under Grant CCR–9504145. Multiprocessor computing facilities were provided through a generous donation by Sun Microsystems.

An earlier version of this paper appeared in the *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998.

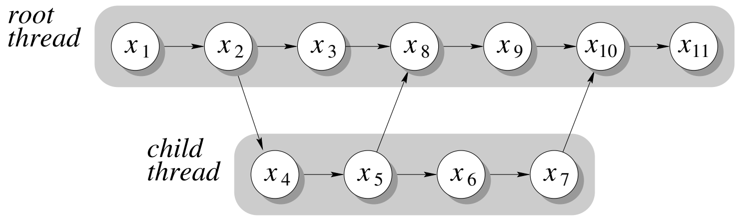


Figure 1: An example computation dag. This dag has 11 nodes $x_1, x_2, \ldots, x_{11}$ and 2 threads indicated by the shading.

In a multiprogrammed environment, a parallel computation runs on a collection of processors that grows and shrinks over time. Initially the computation may be the only one running, and it may use all P processors. A moment later, someone may launch another computation, possibly a serial computation, that runs on some processor. In this case, the parallel computation gives up one processor and continues running on the remaining $P - 1$ processors. Later, if the serial computation terminates or waits for I/O, the parallel computation can resume its use of all processors. In general, other serial and parallel computations may use processors in a time-varying manner that is beyond our control. Thus, we assume that an adversary controls the set of processors on which a parallel computation runs.

Specifically, rather than mapping threads to processors, our thread scheduler maps threads to a fixed collection of P processes, and an adversary maps processes to processors. Throughout this paper, we use the word "process" to denote a kernel-level thread (also called a light-weight process), and we reserve the word "thread" to denote a user-level thread. We model a multiprogrammed environment with two levels of scheduling. A user-level scheduler — our scheduler — maps threads to processes, and below this level, the kernel — an adversary — maps processes to processors. In this environment, we cannot expect to achieve P-fold speedups, because the kernel may run our computation on fewer than P processors. Rather, we let P_A denote the time-average number of processors on which the kernel executes our computation, and we strive to achieve a P_A-fold speedup.

As with much previous work, we model a multithreaded computation as a directed acyclic graph, or ***dag***. An example is shown in Figure 1. Each node in the dag represents a single instruction, and the edges represent ordering constraints. The nodes of a thread are linked by edges that form a chain corresponding to the dynamic instruction execution order of the thread. The example in Figure 1 has two threads indicated by the shaded regions. When an instruction in one thread spawns a new child thread, then the dag has an edge from the "spawning" node in the parent thread to the first node in the new child thread. The edge (x_2, x_4) is such an edge. Likewise, whenever threads synchronize such that an instruction in one thread cannot be executed until after some instruction in another thread, then the dag contains an edge from the node representing the latter instruction to the node representing the former instruction. For example, edge (x_7, x_{10}) represents the joining of the two threads, and edge (x_5, x_8) represents a synchronization that could arise from the use of semaphores [16] — node x_8 represents the P (wait) operation, and node x_5 represents the V (signal) operation on a semaphore whose initial value is 0.

We make two assumptions related to the structure of the dag. First, we assume that each node has out-degree at most 2. This assumption is consistent with our convention that a node represents a single instruction. Second, we assume that the dag has exactly one ***root node*** with in-degree 0 and one ***final node*** with out-degree 0. The root node is the first node of the ***root thread***.

We characterize the computation with two measures: work and critical-path length. The ***work*** T_1 of the computation is the number of nodes in the dag, and the ***critical-path length*** T_∞ is the length of a longest (directed) path in the dag. The ratio T_1/T_∞ is called the ***parallelism***. The example computation of Figure 1 has work $T_1 = 11$, critical-path length $T_\infty = 8$, and parallelism $T_1/T_\infty = 11/8$.

We present a non-blocking implementation of the work-stealing algorithm [8], and we analyze the performance of this non-blocking work stealer in multiprogrammed environments. In this implementation, all concurrent data structures are non-blocking [26, 27] so that if the kernel preempts a process, it does not hinder other processes, for example by holding locks. Moreover, this implementation makes use of "yield" system calls that constrain the kernel adversary in a manner that models the behavior of `yield` system calls found in current multiprocessor operating systems. When a process calls `yield`, it informs the kernel that it wishes to yield the processor on which it is running to another process. Our results demonstrate the surprising power of `yield` as a scheduling primitive. In particular, we show that for any multithreaded computation with work T_1 and critical-path length T_∞, the non-blocking work stealer runs in expected time $O(T_1/P_A + T_\infty P/P_A)$. This bound is optimal to within a constant factor and achieves linear speedup — that is, execution time $O(T_1/P_A)$ — whenever $P = O(T_1/T_\infty)$. We also show that for any $\varepsilon > 0$, with probability at least $1 - \varepsilon$, the execution time is $O(T_1/P_A + (T_\infty + \lg(1/\varepsilon))P/P_A)$.

This result improves on previous results [8] in two ways. First, we consider arbitrary multithreaded computations as opposed to the special case of "fully strict" computations. Second, we consider multiprogrammed environments as opposed to dedicated environments. A multiprogrammed environment is a generalization of a dedicated environment, because we can view a dedicated environment as a multiprogrammed environment in which the kernel executes the computation on P dedicated processors. Moreover, note that in this case, we have $P_A = P$, and our bound for multiprogrammed environments specializes to match the $O(T_1/P + T_\infty)$ bound established earlier for fully strict computations executing in dedicated environments.

Our non-blocking work stealer has been implemented in a prototype `C++` threads library called ***Hood*** [10], and numerous performance studies have been conducted [9, 10]. These studies show that application performance conforms to the $O(T_1/P_A + T_\infty P/P_A)$ bound and that the constant hidden in the big-Oh notation is small, roughly 1. Moreover, these studies show that non-blocking data structures and the use of yields are essential in practice. If any of these implementation mechanisms are omitted, then performance degrades dramatically for $P_A < P$.

The remainder of this paper is organized as follows. In Section 2, we formalize our model of multiprogrammed environments. We also prove a lower bound implying that the performance of the non-blocking work stealer is optimal to within a constant factor. We present the non-blocking work stealer in Section 3, and we prove an important structural lemma that is needed for the analysis. In Section 4 we establish optimal upper bounds on the performance of the work stealer under various assumptions with respect to the kernel. In Section 5, we consider related work. In Section 6 we offer some concluding remarks.

2 Multiprogramming

We model a multiprogrammed environment with a kernel that behaves as an adversary. Whereas a user-level scheduler maps threads onto a fixed collection of P processes, the kernel maps processes onto processors. In this section, we define execution schedules, and we prove upper and lower bounds on the length of execution schedules. These bounds are straightforward and are included primarily to give the reader a better understanding of the model of computation and the central issues that we intend to address. The lower bound demonstrates the optimality of the $O(T_1/P_A + T_\infty P/P_A)$ upper bound that we will establish for our non-blocking work stealer.

The kernel operates in discrete ***steps***, numbered from 1, as follows. At each step i, the kernel chooses any subset of the P processes, and then these chosen processes are allowed to execute a single instruction. We let p_i denote the number of chosen processes, and we say that these p_i processes are ***scheduled*** at step i. The kernel may choose to schedule any number of processes between 0 and P, so $0 \le p_i \le P$. We can view the kernel as producing a ***kernel schedule*** that maps each positive integer to a subset of the processes. That is, a kernel schedule maps each step i to the set of processes that are scheduled at step i, and p_i is the size of

	q_1	q_2	q_3
1	✓	✓	
2	✓	✓	✓
3			
4	✓		✓
5		✓	✓
6	✓	✓	✓
7			✓
8	✓	✓	
9	✓	✓	✓
10		✓	✓

(a) Kernel schedule.

	q_1	q_2	q_3
1	x_1	I	
2	x_2	I	I
3			
4	x_3		x_4
5		I	x_5
6	x_8	x_6	I
7			x_9
8	x_7	I	
9	I	I	x_{10}
10		x_{11}	I

(b) Execution schedule.

Figure 2: An example kernel schedule and an example execution schedule with $P = 3$ processes. **(a)** The first 10 steps of a kernel schedule. Each row represents a time step, and each column represents a process. A check mark in row i and column j indicates that the process q_j is scheduled at step i. **(b)** An execution schedule for the kernel schedule in (a) and the computation dag in Figure 1. The execution schedule shows the activity of each process at each step for which it is scheduled. Each entry is either a node x_i in case the process executes node x_i or "I" in case the process does not execute a node.

that set. The first 10 steps of an example kernel schedule for $P = 3$ processes are shown in Figure 2(a). (In general, kernel schedules are infinite.) The ***processor average*** P_A over T steps is defined as

$$P_A = \frac{1}{T}\sum_{i=1}^{T} p_i \; . \tag{1}$$

In the kernel schedule of Figure 2(a), the processor average over 10 steps is $P_A = 20/10 = 2$.

Though our analysis is based on this step-by-step, synchronous execution model, our work stealer is asynchronous and does not depend on synchrony for correctness. The synchronous model admits the possibility that at a step i, two or more processes may execute instructions that reference a common memory location. We assume that the effect of step i is equivalent to some serial execution of the p_i instructions executed by the p_i scheduled processes, where the order of execution is determined in some arbitrary manner by the kernel.

Given a kernel schedule and a computation dag, an ***execution schedule*** specifies, for each step i, the particular subset of at most p_i ready nodes to be executed by the p_i scheduled processes at step i. We define the ***length*** of an execution schedule to be the number of steps in the schedule. Figure 2(b) shows an example execution schedule for the kernel schedule in Figure 2(a) and the dag in Figure 1. This schedule has length 10. An execution schedule observes the dependencies represented by the dag. That is, every node is executed, and for every edge (u, v), node u is executed at a step prior to the step at which node v is executed.

The following theorem shows that T_1/P_A and $T_\infty P/P_A$ are both lower bounds on the length of any execution schedule. The lower bound of T_1/P_A holds regardless of the kernel schedule, while the lower bound of $T_\infty P/P_A$ holds only for some kernel schedules. That is, there exist kernel schedules such that any execution schedule has length at least $T_\infty P/P_A$. Moreover, there exist such kernel schedules with P_A ranging from P down to values arbitrarily close to 0. These lower bounds imply corresponding lower bounds on the performance of any user-level scheduler.

Theorem 1 *Consider any multithreaded computation with work T_1 and critical-path length T_∞, and any number P of processes. Then for any kernel schedule, every execution schedule has length at least T_1/P_A, where P_A is the processor average over the length of the schedule. In addition, for any number P'_A of the form $T_\infty P/(k + T_\infty)$ where k is a nonnegative integer, there exists a kernel schedule such that every execution schedule has length at least $T_\infty P/P_A$, where P_A is the processor average over the length of the schedule and is in the range $\lfloor P'_A \rfloor \le P_A \le P'_A$.*

Proof: The processor average over the length T of the schedule is defined by Equation (1), so we have

$$T = \frac{1}{P_A} \sum_{i=1}^{T} p_i \ . \tag{2}$$

For both lower bounds, we bound T by bounding $\sum_{i=1}^{T} p_i$. The lower bound of T_1/P_A is immediate from the lower bound $\sum_{i=1}^{T} p_i \ge T_1$, which follows from the fact that any execution schedule is required to execute all of the nodes in the multithreaded computation. For the lower bound of $T_\infty P/P_A$, we prove the lower bound $\sum_{i=1}^{T} p_i \ge T_\infty P$.

We construct a kernel schedule that forces every execution schedule to satisfy this bound as follows. Let k be as defined in the statement of the lemma. The kernel schedule sets $p_i = 0$ for $1 \le i \le k$, sets $p_i = P$ for $k+1 \le i \le k + T_\infty$, and sets $p_i = \lfloor P'_A \rfloor$ for $k + T_\infty < i$. Any execution schedule has length $T \ge k + T_\infty$, so we have the lower bound $\sum_{i=1}^{T} p_i \ge T_\infty P$. It remains only to show that P_A is in the desired range. The processor average for the first $k + T_\infty$ steps is $T_\infty P/(k + T_\infty) = P'_A$. For all subsequent steps $i > k + T_\infty$, we have $p_i = \lfloor P'_A \rfloor$. Thus, P_A falls within the desired range. ■

In the off-line user-level scheduling problem, we are given a kernel schedule and a computation dag, and the goal is to compute an execution schedule with the minimum possible length. Though the related decision problem is NP-complete [37], a factor-of-2 approximation algorithm is quite easy. In particular, for some kernel schedules, any level-by-level (Brent [12]) execution schedule or any "greedy" execution schedule is within a factor of 2 of optimal. In addition, though we shall not prove it, for any kernel schedule, some greedy execution schedule is optimal. We say that an execution schedule is ***greedy*** if at each step i the number of ready nodes executed is equal to the minimum of p_i and the number of ready nodes. The execution schedule in Figure 2(b) is greedy. The following theorem about greedy execution schedules also holds for level-by-level execution schedules, with only trivial changes to the proof.

Theorem 2 (Greedy Schedules) *Consider any multithreaded computation with work T_1 and critical-path length T_∞, any number P of processes, and any kernel schedule. Any greedy execution schedule has length at most $T_1/P_A + T_\infty(P-1)/P_A$, where P_A is the processor average over the length of the schedule.*

Proof: Consider any greedy execution schedule, and let T denote its length. As in the proof of Theorem 1, we bound T by bounding $\sum_{i=1}^{T} p_i$. For each step $i = 1, \ldots, T$, we collect p_i tokens, one from each process that is scheduled at step i, and then we bound the total number of tokens collected. Moreover, we collect the tokens in two buckets: a ***work bucket*** and an ***idle bucket***. Consider a step i and a process that is scheduled at step i. If the process executes a node of the computation, then it puts its token into the work bucket, and otherwise we say that the process is idle and it puts its token into the idle bucket. After the last step, the work bucket contains exactly T_1 tokens — one token for each node of the computation. It remains only to prove that the idle bucket contains at most $T_\infty(P-1)$ tokens.

Consider a step during which some process places a token in the idle bucket. We refer to such a step as an ***idle step***. For example, the greedy execution schedule of Figure 2(b) has 7 idle steps. At an idle step we have an idle process and since the schedule is greedy, it follows that every ready node is executed at an idle step. This observation leads to two further observations. First, at every step there is at least one ready node, so of the p_i processes scheduled at an idle step i, at most $p_i - 1 \leq P - 1$ could be idle. Second, for each step i, let G_i denote the sub-dag of the computation consisting of just those nodes that have not yet been executed after step i. If step i is an idle step, then every node with in-degree 0 in G_{i-1} gets executed at step i, so a longest path in G_i is one node shorter than a longest path in G_{i-1}. Since the longest path in G_0 has length T_∞, there can be at most T_∞ idle steps. Putting these two observations together, we conclude that after the last step, the idle bucket contains at most $T_\infty(P - 1)$ tokens. ■

The concern of this paper is on-line user-level scheduling, and an on-line user-level scheduler cannot always produce greedy execution schedules. In the on-line user-level scheduling problem, at each step i, we know the kernel schedule only up through step i, and we know of only those nodes in the dag that are ready or have previously been executed. Moreover, in analyzing the performance of on-line user-level schedulers, we need to account for scheduling overheads. Nevertheless, even though it is an on-line scheduler, and even accounting for all of its overhead, the non-blocking work stealer satisfies the same bound, to within a constant factor, as was shown in Theorem 2 for greedy execution schedules.

3 Non-blocking work stealing

In this section we describe our non-blocking implementation of the work-stealing algorithm. We first review the work-stealing algorithm [8], and then we describe our non-blocking implementation, which involves the use of a yield system call and a non-blocking implementation of the concurrent data structures. We conclude this section with an important "structural lemma" that is used in our analysis.

3.1 The work-stealing algorithm

In the work-stealing algorithm, each process maintains its own pool of ready threads from which it obtains work. A node in the computation dag is ***ready*** if all of its ancestors have been executed, and correspondingly, a thread is ready if it contains a ready node. Note that because all of the nodes in a thread are totally ordered, a thread can have at most one ready node at a time. A ready thread's ready node represents the next instruction to be executed by that thread, as determined by the current value of that thread's program counter. Each pool of ready threads is maintained as a double-ended queue, or ***deque***, which has a bottom and a top. A deque contains only ready threads. If the deque of a process becomes empty, that process becomes a thief and steals a thread from the deque of a victim process chosen at random.

To obtain work, a process pops the ready thread from the bottom of its deque and commences executing that thread, starting with that thread's ready node and continuing in sequence, as determined by the control flow of the code being executed by that thread. We refer to the thread that a process is executing as the process's ***assigned thread***. The process continues to execute nodes in its assigned thread until that thread invokes a synchronization action (typically via a call into the threads library). The synchronization actions fall into the following four categories, and they are handled as follows.

- ***Die:*** When the process executes its assigned thread's last node, that thread dies. In this case, the process gets a new assigned thread by popping one off the bottom of its deque.
- ***Block:*** If the process reaches a node in its assigned thread that is not ready, then that thread blocks. For example, consider a process that is executing the root thread of Figure 1. If the process executes x_3 and then goes to execute x_8 before node x_5 has been executed, then the root thread blocks. In this case, as in the case of the thread dying, the process gets a new assigned thread by popping one off the bottom of its deque.
- ***Enable:*** If the process executes a node in its assigned thread that causes another thread — a thread that previously was blocked — to be ready, then, of the two ready threads (the assigned thread and

the newly ready thread), the process pushes one onto the bottom of its deque and continues executing the other. That other thread becomes the process's assigned thread. For example, if the root thread of Figure 1 is blocked at x_8, waiting for x_5 to be executed, then when a process that is executing the child thread finally executes x_5, the root thread becomes ready and the process performs one of the following two actions. Either it pushes the root thread on the bottom of its deque and continues executing the child thread at x_6, or it pushes the child thread on the bottom of its deque and starts executing the root thread at x_8. The bounds proven in this paper hold for either choice.

- ***Spawn:*** If the process executes a node in its assigned thread that spawns a child thread, then, as in the enabling case, of the two ready threads (in this case, the assigned thread and its newly spawned child), the process pushes one onto the bottom of its deque and continues executing the other. That other thread becomes the process's assigned thread. For example, when a process that is executing the root thread of Figure 1 executes x_2, the process performs one of the following two actions. Either it pushes the child thread on the bottom of its deque and continues executing the root thread at x_3, or it pushes the root thread on the bottom of its deque and starts executing the child thread at x_4. The bounds proven in this paper hold for either choice. The latter choice is often used [21, 22, 31], because it follows the natural depth-first single-processor execution order.

It is possible that a thread may enable another thread and die simultaneously. An example is the join between the root thread and the child thread in Figure 1. If the root thread is blocked at x_{10}, then when a process executes x_7 in the child, the child enables the root and dies simultaneously. In this case, the root thread becomes the process's new assigned thread, and the process commences executing the root thread at x_{10}. Effectively, the process performs the action for enabling followed by the action for dying.

When a process goes to get an assigned thread by popping one off the bottom of its deque, if it finds that its deque is empty, then the process becomes a ***thief***. It picks a ***victim*** process at random (using a uniform distribution) and attempts to steal a thread from the victim by popping a thread off the top of the victim's deque. The steal attempt will fail if the victim's deque is empty. In addition, the steal attempt may fail due to contention when multiple thieves attempt to steal from the same victim simultaneously. The next two sections cover this issue in detail. If the steal attempt fails, then the thief picks another victim process and tries again. The thief repeatedly attempts to steal from randomly chosen victims until it succeeds, at which point the thief "reforms" (i.e., ceases to be a thief). The stolen thread becomes the process's new assigned thread, and the process commences executing its new assigned thread, as described above.

In our non-blocking implementation of the work-stealing algorithm, each process performs a yield system call between every pair of consecutive steal attempts. We describe the semantics of the yield system call later in Section 4.4. These system calls are not needed for correctness, but as we shall see in Section 4.4, the yields are sometimes needed in order to prevent the kernel from starving a process.

Execution begins with all deques empty and the root thread assigned to one process. This one process begins by executing its assigned thread, starting with the root node. All other processes begin as thieves. Execution ends when some process executes the final node, which sets a global flag, thereby terminating the scheduling loop.

For our analysis, we ignore threads. We treat the deques as if they contain ready nodes instead of ready threads, and we treat the scheduler as if it operates on nodes instead of threads. In particular, we replace each ready thread in a deque with its currently ready node. In addition, if a process has an assigned thread, then we define the process's ***assigned node*** to be the currently ready node of its assigned thread.

The scheduler operates as shown in Figure 3. The root node is assigned to one process, and all other processes start with no assigned node (lines 1 through 3). These other processes will become thieves. Each process executes the scheduling loop, which terminates when some process executes the final node and sets a global flag (line 4). At each iteration of the scheduling loop, each process performs as follows.

```
// Assign root node to process zero.
assignedNode ← NIL
if self = processZero
    assignedNode ← rootNode

// Run scheduling loop.
while computationDone = FALSE

    // Execute assigned node.
    if assignedNode ≠ NIL
        (numChildren, child1, child2) ← execute (assignedNode)

        if numChildren = 0                     // Terminate or block.
            assignedNode ← self.popBottom()
        else if numChildren = 1                // No synchronization.
            assignedNode ← child1
        else                                   // Enable or spawn.
            self.pushBottom (child1)
            assignedNode ← child2

    // Make steal attempt.
    else
        yield()                                // Yield processor.
        victim ← randomProcess()               // Pick victim.
        assignedNode ← victim.popTop()         // Attempt steal.
```

Figure 3: The non-blocking work stealer. All P processes execute this scheduling loop. Each process is represented by a `Process` data structure, stored in shared memory, that contains the deque of the process, and each process has a private variable `self` that refers to its `Process` structure. Initially, all deques are empty and the `computationDone` flag, which is stored in shared memory, is `FALSE`. The root node is assigned to an arbitrary process, designated `processZero`, prior to entering the main scheduling loop. The scheduling loop terminates when a process executes the final node and sets the `computationDone` flag.

If the process has an assigned node, then it executes that assigned node (lines 5 and 6). The execution of the assigned node will enable — that is, make ready — 0, 1, or 2 child nodes. Specifically, it will enable 0 children in case the assigned thread dies or blocks; it will enable 1 child in case the assigned thread performs no synchronization, merely advancing to the next node; and it will enable 2 children in case the assigned thread enables another, previously blocked, thread or spawns a child thread. If the execution of the assigned node enables 0 children, then the process pops the ready node off the bottom of its deque, and this node becomes the process's new assigned node (lines 7 and 8). If the process's deque is empty, then the pop invocation returns `NIL`, so the process does not get a new assigned node and becomes a thief. If the execution of the assigned node enables 1 child, then this child becomes the process's new assigned node (lines 9 and 10). If the the execution of the assigned node enables 2 children, then the process pushes one of the children onto the bottom of its deque, and the other child becomes the process's new assigned node (lines 11 through 13).

If a process has no assigned node, then its deque is empty, so it becomes a thief. The thief picks a victim at random and attempts to pop a node off the top of the victim's deque, making that node its new assigned node (lines 16 and 17). If the steal attempt is unsuccessful, then the pop invocation returns `NIL`, so the thief does not get an assigned node and continues to be a thief. If the steal attempt is successful, then the pop invocation returns a node, so the thief gets an assigned node and reforms. Between consecutive steal attempts, the thief calls `yield` (line 15).

3.2 Specification of the deque methods

In this section we develop a specification for the deque object, discussed informally above. The deque supports three methods: `pushBottom`, `popBottom`, and `popTop`. A `pushTop` method is not supported, because it is not needed by the work-stealing algorithm. A deque implementation is defined to be ***constant-time*** if and only if each of the three methods terminates within a constant number of instructions. Below we define the "ideal" semantics of these methods. Any constant-time deque implementation meeting the ideal semantics is wait-free [27]. Unfortunately, we are not aware of any constant-time wait-free deque implementation. For this reason, we go on to define a "relaxed" semantics for the deque methods. Any constant-time deque implementation meeting the relaxed semantics is non-blocking [26, 27] and is sufficient for us to prove our performance bounds.

We now define the ideal deque semantics. To do so, we first define whether a given set of invocations of the deque methods meets the ideal semantics. We view an invocation of a deque method as a 4-tuple specifying: (i) the name of the deque method invoked (i.e., `pushBottom`, `popBottom`, or `popTop`), (ii) the initiation time, (iii) the completion time, and (iv) the argument (for the case of `pushBottom`) or the return value (for `popBottom` and `popTop`). A set of invocations meets the ideal semantics if and only if there exists a ***linearization time*** for each invocation such that: (i) the linearization time lies between the initiation time and the completion time, (ii) no two linearization times coincide, and (iii) the return values are consistent with a serial execution of the method invocations in the order given by the linearization times. A deque implementation meets the ideal semantics if and only if for any execution, the associated set of invocations meets the ideal semantics. We remark that a deque implementation meets the ideal semantics if and only if each of the three deque methods is *linearizable*, as defined in [25].

It is convenient to define a set of invocations to be ***good*** if and only if no two `pushBottom` or `popBottom` invocations are concurrent. Note that any set of invocations associated with some execution of the work-stealing algorithm is good since the (unique) owner of each deque is the only process to ever perform either a `pushBottom` or `popBottom` on that deque. Thus, for present purposes, it is sufficient to design a constant-time wait-free deque implementation that meets the ideal semantics on any good set of invocations. Unfortunately, we do not know how to do this. On the positive side, we are able to establish optimal performance bounds for the work-stealing algorithm even if the deque implementation satisfies only a relaxed version of the ideal semantics.

In the relaxed semantics, we allow a `popTop` invocation to return `NIL` if at some point during the invocation, either the deque is empty (this is the usual condition for returning `NIL`) or the topmost item is removed from the deque by another process. In the next section we provide a constant-time non-blocking deque implementation that meets the relaxed semantics on any good set of invocations. We do not consider our implementation to be wait-free, because we do not view every `popTop` invocation that returns `NIL` as having successfully completed. Specifically, we consider a `popTop` invocation that returns `NIL` to be successful if and only if the deque is empty at some point during the invocation. Note that a successful `popTop` invocation is linearizable.

3.3 The deque implementation

The deques support concurrent method invocations, and we implement the deques using non-blocking synchronization. Such an implementation requires the use of a universal primitive such as compare-and-swap or load-linked/store-conditional [27]. Almost all modern microprocessors have such instructions. In our deque implementation we employ a compare-and-swap instruction, but this instruction can be replaced with a load-linked/store-conditional pair in a straightforward manner [32].

The compare-and-swap instruction `cas` operates as follows. It takes three operands: a register `addr` that holds an address and two other registers, `old` and `new`, holding arbitrary values. The instruction `cas (addr, old, new)` compares the value stored in memory location `addr` with `old`, and if they are equal, the value stored in memory location `addr` is swapped with `new`. In this case, we say the `cas` ***succeeds***. Otherwise, it loads the value stored in memory location `addr` into `new`, without modifying the memory location `addr`. In this case, we say the `cas` ***fails***. This whole operation — comparing and then either swapping or loading — is performed atomically with respect to all other memory operations. We can detect whether the `cas` fails or succeeds by comparing `old` with `new` after the `cas`. If they are equal, then the `cas` succeeded; otherwise, it failed.

In order to implement a deque of nodes (or threads) in a non-blocking manner using `cas`, we employ an array of nodes (or pointers to threads), and we store the indices of the top and bottom entries in the variables `top` and `bot` respectively, as shown in Figure 4. An additional variable `tag` is required for correct operation, as described below. The `tag` and `top` variables are implemented as fields of a structure `age`, and this structure is assumed to fit within a single word, which we define as the maximum number of bits that can be transfered to and from memory atomically with `load`, `store`, and `cas` instructions. The `age` structure fits easily within either a 32-bit or a 64-bit word size.

The tag field is needed to address the following potential problem. Suppose that a thief process is preempted after executing line 5 but before executing line 8 of `popTop`. Subsequent operations may empty the deque and then build it up again so that the top index points to the same location. When the thief process resumes and executes line 8, the `cas` will succeed because the top index has been restored to its previous value. But the node that the thief obtained at line 5 is no longer the correct node. The tag field eliminates this problem, because every time the top index is reset (line 11 of `popBottom`), the tag is changed. This changing of the tag will cause the thief's `cas` to fail. For simplicity, in Figure 5 we show the tag being manipulated as a counter, with a new tag being selected by incrementing the old tag (line 12 of `popBottom`). Such a tag might wrap around, so in practice, we implement the tag by adapting the "bounded tags" algorithm [32].

We claim that the deque implementation presented above meets the relaxed semantics on any good set of invocations. Even though each of the deque methods is loop-free and consists of a relatively small number of instructions, proving this claim is not entirely trivial since we need to account for every possible interleaving of the executions of the owner and thieves. Our current proof of correctness is somewhat lengthy as it reduces the problem to establishing the correctness of a rather large number of sequential

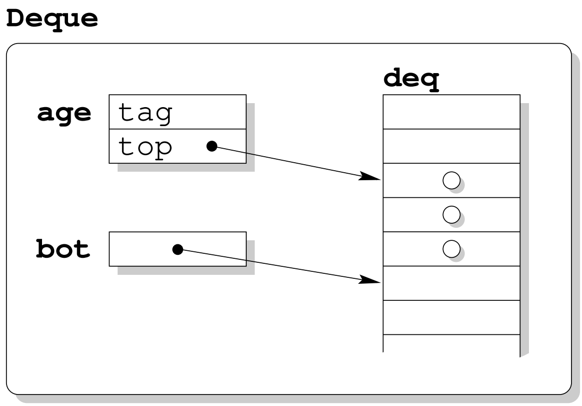


Figure 4: A Deque object contains an array deq of ready nodes, a variable bot that is the index below the bottom node, and a variable age that contains two fields: top, the index of the top node, and tag, a "uniquifier" needed to ensure correct operation. The variable age fits in a single word of memory that can be operated on with atomic load, store, and cas instructions.

```
void pushBottom (Node node)
load localBot ← bot
store node → deq[localBot]
localBot ← localBot + 1
store localBot → bot
```

```
Node popTop()
load oldAge ← age
load localBot ← bot
if localBot ≤ oldAge.top
    return NIL
load node ← deq[oldAge.top]
newAge ← oldAge
newAge.top ← newAge.top + 1
cas (age, oldAge, newAge)
if oldAge = newAge
    return node
return NIL
```

```
Node popBottom()
load localBot ← bot
if localBot = 0
    return NIL
localBot ← localBot − 1
store localBot → bot
load node ← deq[localBot]
load oldAge ← age
if localBot > oldAge.top
    return node
store 0 → bot
newAge.top ← 0
newAge.tag ← oldAge.tag + 1
if localBot = oldAge.top
    cas (age, oldAge, newAge)
    if oldAge = newAge
        return node
store newAge → age
return NIL
```

Figure 5: The three Deque methods. Each Deque object resides in shared memory along with its instance variables age, bot, and deq; the remaining variables in this code are private (registers). The load, store, and cas instructions operate atomically. On a multiprocessor that does not support sequential consistency, extra memory operation ordering instructions may be needed.

program fragments. Because program verification is not the primary focus of the present article, the proof of correctness is omitted. The reader interested in program verification is referred to [11] for a detailed presentation of the correctness proof.

The fact that our deque implementation meets the relaxed semantics on any good set of invocations greatly simplifies the performance analysis of the work-stealing algorithm. For example, by ensuring the linearizability of all owner invocations and all thief invocations that do not return `NIL`, this fact allows us to view such invocations as atomic. Under this view, the precise state of the deque at any given point in the execution has a clear definition in terms of the usual serial semantics of the deque methods `pushBottom`, `popBottom`, and `popTop`. (Here we rely on the observation that a thief invocation returning `NIL` does not change the state of the shared memory, and hence does not change the state of the deque.)

3.4 A structural lemma

In this section we establish a key lemma that is used in the performance analysis of our work-stealing scheduler. Before stating the lemma, we provide a number of technical definitions.

To state the structural lemma, in addition to linearizing the deque method invocations as described in the previous section, we also need to linearize the assigned-node executions. If the execution of the assigned node enables 0 children, then we view the execution and subsequent updating of the assigned node as occurring atomically at the linearization point of the ensuing `popBottom` invocation. If the execution of the assigned node enables 1 child, then we view the execution and updating of the assigned node as occurring atomically at the time the assigned node is executed. If the execution of the assigned node enables 2 children, then we view the execution and updating of the assigned node as occurring atomically at the linearization point of the ensuing `pushBottom` invocation. In each of the above cases, the choice of linearization point is justified by the following simple observation: the execution of any local instruction (i.e., an instruction that does not involve the shared memory) by some process commutes with the execution of any instruction by another process.

If the execution of node u enables node v, then we call the edge (u, v) an ***enabling edge***, and we call u the ***designated parent*** of v. Note that every node except the root node has exactly one designated parent, so the subgraph of the dag consisting of only enabling edges forms a rooted tree that we call the ***enabling tree***. Note that each execution of the computation may have a different enabling tree. If $d(u)$ is the depth of a node u in the enabling tree, then its ***weight*** is defined as $w(u) = T_\infty - d(u)$. The root of the dag, which is also the root of the enabling tree, has weight T_∞. Our analysis of Section 4 employs a potential function based on the node weights.

As illustrated in Figure 6, the structural lemma states that for any deque, at all times during the execution of the work-stealing algorithm, the designated parents of the nodes in the deque lie on some root-to-leaf path in the enabling tree. Moreover, the ordering of these designated parents along this path corresponds to the top-to-bottom ordering of the nodes in the deque. As a corollary, we observe that the weights of the nodes in the deque are strictly decreasing from top to bottom.

Lemma 3 (Structural Lemma) *Let k be the number of nodes in a given deque at some time in the (linearized) execution of the work-stealing algorithm, and let $v_1, \ldots, v_k$ denote those nodes ordered from the bottom of the deque to the top. Let v_0 denote the assigned node if there is one. In addition, for $i = 0, \ldots, k$, let u_i denote the designated parent of v_i. Then for $i = 1, \ldots, k$, node u_i is an ancestor of u_{i-1} in the enabling tree. Moreover, though we may have $u_1 = u_0$, for $i = 2, 3, \ldots, k$, we have $u_i \neq u_{i-1}$ — that is, the ancestor relationship is proper.*

Proof: Fix a particular deque. The deque state and assigned node change only when either the owner executes its assigned node or a thief performs a successful steal. We prove the claim by induction on the

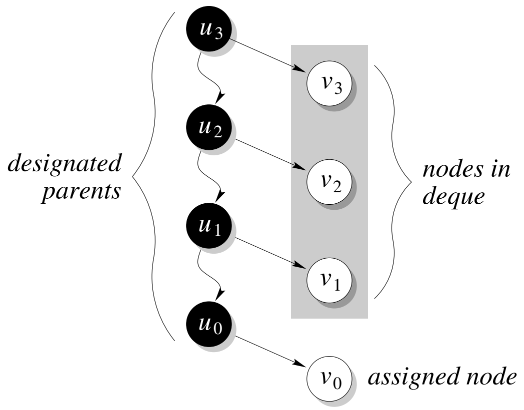
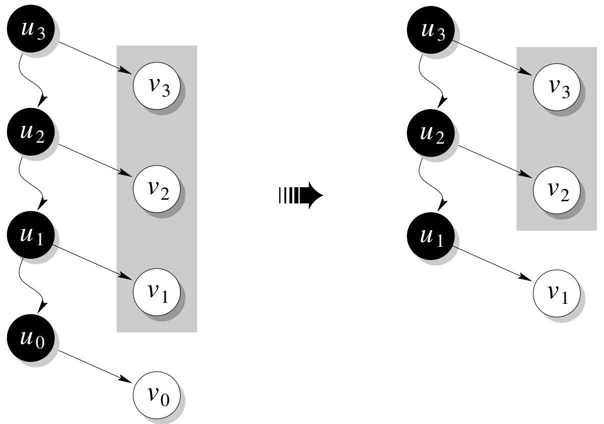


Figure 6: The structure of the nodes in the deque of some process. Node v_0 is the assigned node. Nodes v_1, v_2, and v_3 are the nodes in the deque ordered from bottom to top. For $i = 0, 1, 2, 3$, node u_i is the designated parent of node v_i. Then nodes u_3, u_2, u_1, and u_0 lie (in that order) on a root-to-leaf path in the enabling tree. As indicated in the statement of Lemma 3, the u_i's are all distinct except it is possible that $u_0 = u_1$.



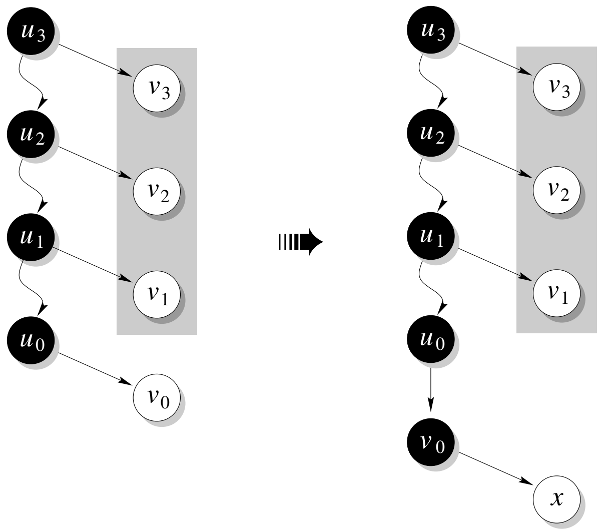
(a) Before. **(b)** After.

Figure 7: The deque of a processor before and after the execution of the assigned node v_0 enables 0 children.

number of assigned-node executions and steals since the deque was last empty. In the base case, if the deque is empty, then the claim holds vacuously. We now assume that the claim holds before a given assigned-node execution or successful steal, and we will show that it holds after. Specifically, before the assigned-node execution or successful steal, let v_0 denote the assigned node; let k denote the number of nodes in the deque; let $v_1, \ldots, v_k$ denote the nodes in the deque ordered from bottom to top; and for $i = 0, \ldots, k$, let u_i denote the designated parent of v_i. We assume that either $k = 0$, or for $i = 1, \ldots, k$, node u_i is an ancestor of u_{i-1} in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$. After the assigned-node execution or successful steal, let v'_0 denote the assigned node; let k' denote the number of nodes in the deque; let $v'_1, \ldots, v'_k$ denote the nodes in the deque ordered from bottom to top; and for $i = 0, \ldots, k'$, let u'_i denote the designated parent of v'_i. We now show that either $k' = 0$, or for $i = 1, \ldots, k'$, node u'_i is an ancestor of u'_{i-1} in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$.

Consider the execution of the assigned node v_0 by the owner.

If the execution of v_0 enables 0 children, then the owner pops the bottommost node off its deque and makes that node its new assigned node. If $k = 0$, then the deque is empty; the owner does not get a new



(a) Before. **(b)** After.

Figure 8: The deque of a processor before and after the execution of the assigned node v_0 enables 1 child x.

assigned node; and $k' = 0$. If $k > 0$, then the bottommost node v_1 is popped and becomes the new assigned node, and $k' = k - 1$. If $k = 1$, then $k' = 0$. Otherwise, the result is as illustrated in Figure 7. We now rename the nodes as follows. For $i = 0, \ldots, k'$, we set $v'_i = v_{i+1}$ and $u'_i = u_{i+1}$. We now observe that for $i = 1, \ldots, k'$, node u'_i is a proper ancestor of u'_{i-1} in the enabling tree.

If the execution of v_0 enables 1 child x, then, as illustrated in Figure 8, x becomes the new assigned node; the designated parent of x is v_0; and $k' = k$. If $k = 0$, then $k' = 0$. Otherwise, we can rename the nodes as follows. We set $v'_0 = x$; we set $u'_0 = v_0$; and for $i = 1, \ldots, k'$, we set $v'_i = v_i$ and $u'_i = u_i$. We now observe that for $i = 1, \ldots, k'$, node u'_i is a proper ancestor of u'_{i-1} in the enabling tree. That u'_1 is a proper ancestor of u'_0 in the enabling tree follows from the fact that (u_0, v_0) is an enabling edge.

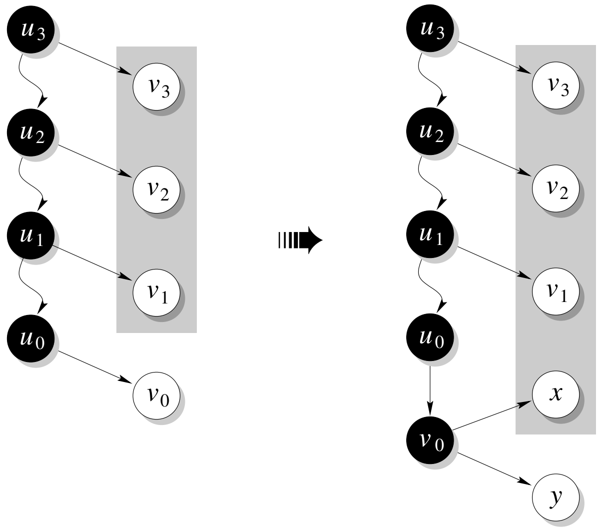
In the most interesting case, the execution of the assigned node v_0 enables 2 children x and y, with x being pushed onto the bottom of the deque and y becoming the new assigned node, as illustrated in Figure 9. In this case, (v_0, x) and (v_0, y) are both enabling edges, and $k' = k + 1$. We now rename the nodes as follows. We set $v'_0 = y$; we set $u'_0 = v_0$; we set $v'_1 = x$; we set $u'_1 = v_0$; and for $i = 2, \ldots, k'$, we set $v'_i = v_{i-1}$ and $u'_i = u_{i-1}$. We now observe that $u'_1 = u'_0$, and for $i = 2, \ldots, k'$, node u'_i is a proper ancestor of u'_{i-1} in the enabling tree. That u'_2 is a proper ancestor of u'_1 in the enabling tree follows from the fact that (u_0, v_0) is an enabling edge.

Finally, we consider a successful steal by a thief. In this case, the thief pops the topmost node v_k off the deque, so $k' = k - 1$. If $k = 1$, then $k' = 0$. Otherwise, we can rename the nodes as follows. For $i = 0, \ldots, k'$, we set $v'_i = v_i$ and $u'_i = u_i$. We now observe that for $i = 1, \ldots, k'$, node u'_i is an ancestor of u'_{i-1} in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$. ■

Corollary 4 *If $v_0, v_1, \ldots, v_k$ are as defined in the statement of Lemma 3, then we have $w(v_0) \leq w(v_1) < \cdots < w(v_{k-1}) < w(v_k)$.* ■

4 Analysis of the work stealer

In this section we establish optimal bounds on the running time of the non-blocking work stealer under various assumptions about the kernel. It should be emphasized that the work stealer performs correctly for



(a) Before. **(b)** After.

Figure 9: The deque of a processor before and after the execution of the assigned node v_0 enables 2 children x and y.

any kernel. We consider various restrictions on kernel behavior in order to demonstrate environments in which the running time of the work stealer is optimal.

The following definitions will prove to be useful in our analysis. An instruction in the sequence executed by some process q is a ***milestone*** if and only if one of the following two conditions holds: (i) execution of a node by process q occurs at that instruction, or (ii) a `popTop` invocation completes. From the scheduling loop of Figure 3, we observe that a given process may execute at most some constant number of instructions between successive milestones. Throughout this section, we let C denote a sufficiently large constant such that in any sequence of C consecutive instructions executed by a process, at least one is a milestone.

The remainder of this section is organized as follows. Section 4.1 reduces the analysis to bounding the number of "throws". Section 4.2 defines a potential function that is central to all of our upper-bound arguments. Sections 4.3 and 4.4 present our upper bounds for dedicated and multiprogrammed environments.

4.1 Throws

In this section we show that the execution time of our work stealer is $O(T_1/P_A + S/P_A)$, where S is the number of "throws", that is, steal attempts satisfying a technical condition stated below. This goal cannot be achieved without restricting the kernel, so in addition to proving this bound on execution time, we shall state and justify certain kernel restrictions.

One fundamental obstacle prevents us from proving the desired performance bound within the (unrestricted) multiprogramming model of Section 2. The problem is that the kernel may bias the random steal attempts towards the empty deques. In particular, consider the steal attempts initiated within some fixed interval of steps. The adversary can bias these steal attempts towards the empty deques by delaying those steal attempts that choose nonempty deques as victims so that they occur after the end of the interval.

To address this issue, we restrict the kernel to schedule in ***rounds*** rather than steps. A process that is scheduled in a particular round executes between $2C$ and $3C$ instructions during the round, where C is the constant defined at the beginning of Section 4. The precise number of instructions that a process executes during a round is determined by the kernel in an arbitrary manner. We assume that the process executes these $2C$ to $3C$ instructions in serial order, but we allow the instruction streams of different processes to be interleaved arbitrarily, as determined by the kernel. We claim that our requirement that processes be scheduled in rounds of $2C$ to $3C$ instructions is a reasonable one. Because of the overhead associated with context-switching, practical kernels tend to assign processes to processors for some nontrivial scheduling quantum. In fact, a typical scheduling quantum is orders of magnitude higher than the modest value of C needed to achieve our performance bounds.

We identify the completion of a steal attempt with the completion of its `popTop` invocation (line 17 of the scheduling loop), and we define a steal attempt by a process q to be a ***throw*** if it completes at q's second milestone in a round. Thus a process performs at most one throw in any round. Such a throw completes in the round in which the identity of the associated random victim is determined. This property is useful because it ensures that the random victim distribution cannot be biased by the kernel. The following lemma bounds the execution time in terms of the number of throws.

Lemma 5 *Consider any multithreaded computation with work T_1 being executed by the non-blocking work stealer. Then the execution time is at most $O(T_1/P_A + S/P_A)$, where S denotes the number of throws.*

Proof: As in the proof of Theorem 2, we bound the execution time by using Equation (2) and bounding $\sum_{i=1}^{T} p_i$. At each round, we collect a token from each scheduled process. We will show that the total number of tokens collected is at most $T_1 + S$. Since each round consists of at most $3C$ steps, this bound on the number of tokens implies the desired time bound.

When a process q is scheduled in a round, it executes at least two milestones, and the process places its token in one of two buckets, as determined by the second milestone. There are two types of milestones. If q's second milestone marks the occurrence of a node execution, then q places its token in the ***work bucket***. Clearly there are at most T_1 tokens in the work bucket. The second type of milestone marks the completion of a steal attempt, and if q's second milestone is of this type, then q places its token in the ***steal bucket***. In this case, we observe that the steal attempt is a throw, so there are exactly S tokens in the steal bucket. ■

4.2 The potential function

As argued in the previous section, it remains only to analyze the number of throws. We perform this analysis using an amortization argument based on a potential function that decreases as the algorithm progresses. Our high-level strategy is to divide the execution into phases and show that in each phase the potential decreases by at least a constant fraction with constant probability.

We define the potential function in terms of node weights. Recall that each node u has a weight $w(u) = T_\infty - d(u)$, where $d(u)$ is the depth of node u in the enabling tree. At any given round i, we define the potential by assigning potential to each ready node. Let R_i denote the set of ready nodes at the beginning of round i. A ready node is either assigned to a process or it is in the deque of some process. For each ready node u in R_i, we define the associated potential $\phi_i(u)$ as

$$\phi_i(u) = \begin{cases} 3^{2w(u)-1} & \text{if } u \text{ is assigned;} \\ 3^{2w(u)} & \text{otherwise.} \end{cases}$$

Then the potential at round i is defined as

$$\Phi_i = \sum_{u \in R_i} \phi_i(u) \; .$$

When execution begins, the only ready node is the root node, which has weight T_∞ and is assigned to some process, so we start with $\Phi_0 = 3^{2T_\infty - 1}$. When execution terminates, there are no ready nodes, so the final potential is 0.

Throughout the execution, the potential never increases. That is, for each round i, we have $\Phi_{i+1} \le \Phi_i$. The work stealer performs only two actions that may change the potential, and both of them decrease the potential. The first action that changes the potential is the removal of a node u from a deque when u is assigned to a process (lines 8 and 17 of the scheduling loop). In this case, the potential decreases by $\phi_i(u) - \phi_{i+1}(u) = 3^{2w(u)} - 3^{2w(u)-1} = (2/3)\phi_i(u)$, which is positive. The second action that changes the potential is the execution of an assigned node u. If the execution of u enables two children, then one child x is placed in the deque and the other y becomes the assigned node. Thus, the potential decreases by

$$\begin{aligned}
&\phi_i(u) - \phi_{i+1}(x) - \phi_{i+1}(y) \\
&\quad = \quad 3^{2w(u)-1} - 3^{2w(x)} - 3^{2w(y)-1} \\
&\quad = \quad 3^{2w(u)-1} - 3^{2(w(u)-1)} - 3^{2(w(u)-1)-1} \\
&\quad = \quad 3^{2w(u)-1}\left(1 - \frac{1}{3} - \frac{1}{9}\right) \\
&\quad = \quad \frac{5}{9}\phi_i(u) \ ,
\end{aligned}$$

which is positive. If the execution of u enables fewer than two children, then the potential decreases even more. Thus, the execution of a node u at round i decreases the potential by at least $(5/9)\phi_i(u)$.

To facilitate the analysis, we partition the potential among the processes, and we separately consider the processes whose deque is empty and the processes whose deque is nonempty. At the beginning of round i, for any process q, let $R_i(q)$ denote the set of ready nodes that are in q's deque along with the ready node, if any, that is assigned to q. We say that each node u in $R_i(q)$ *belongs* to process q. Then the potential that we associate with q is

$$\Phi_i(q) = \sum_{u \in R_i(q)} \phi_i(u) \ .$$

In addition, let A_i denote the set of processes whose deque is empty at the beginning of round i, and let D_i denote the set of all other processes. We partition the potential Φ_i into two parts

$$\Phi_i = \Phi_i(A_i) + \Phi_i(D_i) \ ,$$

where

$$\Phi_i(A_i) = \sum_{q \in A_i} \Phi_i(q) \qquad \text{and} \qquad \Phi_i(D_i) = \sum_{q \in D_i} \Phi_i(q) \ ,$$

and we analyze the two parts separately.

We now wish to show that whenever P or more throws take place over a sequence of rounds, the potential decreases by a constant fraction with constant probability. We prove this claim in two stages. First, we show that $3/4$ of the potential $\Phi_i(D_i)$ is sitting "exposed" at the top of the deques where it is accessible to steal attempts. Second, we use a "balls and weighted bins" argument to show that $1/2$ of this exposed potential is stolen with $1/4$ probability. The potential $\Phi_i(A_i)$ is considered separately.

Lemma 6 (Top-Heavy Deques) *Consider any round i and any process q in D_i. The topmost node u in q's deque contributes at least $3/4$ of the potential associated with q. That is, we have $\phi_i(u) \geq (3/4)\Phi_i(q)$.*

Proof: This lemma follows directly from the Structural Lemma (Lemma 3), and in particular from Corollary 4. Suppose the topmost node u in q's deque is also the only node in q's deque, and in addition, u has the same designated parent as the node y that is assigned to q. In this case, we have

$$\begin{aligned}
\Phi_i(q) &= \phi_i(u) + \phi_i(y) \\
&= 3^{2w(u)} + 3^{2w(y)-1} \\
&= 3^{2w(u)} + 3^{2w(u)-1} \\
&= \frac{4}{3}\phi_i(u) \ .
\end{aligned}$$

In all other cases, u contributes an even larger fraction of the potential associated with q. ■

Lemma 7 (Balls and Weighted Bins) *Suppose that P balls are thrown independently and uniformly at random into P bins, where for $i = 1, \ldots, P$, bin i has a weight W_i. The total weight is $W = \sum_{i=1}^{P} W_i$. For each bin i, define the random variable X_i as*

$$X_i = \begin{cases} W_i & \textit{if some ball lands in bin } i; \\ 0 & \textit{otherwise.} \end{cases}$$

If $X = \sum_{i=1}^{P} X_i$, then for any β in the range $0 < \beta < 1$, we have $\Pr\{X \geq \beta W\} > 1 - 1/((1-\beta)e)$.

Proof: For each bin i, consider the random variable $W_i - X_i$. It takes on the value W_i when no ball lands in bin i, and otherwise it is 0. Thus, we have

$$\begin{aligned} \mathrm{E}[W_i - X_i] &= W_i \left(1 - \frac{1}{P}\right)^P \\ &\leq W_i/e\ . \end{aligned}$$

It follows that $\mathrm{E}[W - X] \leq W/e$. From Markov's Inequality we have that

$$\Pr\{W - X > (1-\beta)W\} < \frac{\mathrm{E}[W-X]}{(1-\beta)W}\ .$$

Thus, we conclude $\Pr\{X < \beta W\} < 1/((1-\beta)e)$. ■

We now show that whenever P or more throws occur, the potential decreases by a constant fraction of $\Phi_i(D_i)$ with constant probability.

Lemma 8 *Consider any round i and any later round j such that at least P throws occur at rounds from i (inclusive) to j (exclusive). Then we have*

$$\Pr\left\{\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i(D_i)\right\} > \frac{1}{4}\ .$$

Proof: We first use the Top-Heavy Deques Lemma to show that if a throw targets a process with a nonempty deque as its victim, then the potential decreases by at least $1/2$ of the potential associated with that victim process. We then consider the P throws as ball tosses, and we use the Balls and Weighted Bins Lemma to show that with probability more than $1/4$, the total potential decreases by $1/4$ of the potential associated with all processes with a nonempty deque.

Consider any process q in D_i, and let u denote the node at the top of q's deque at round i. From the Top-Heavy Deques Lemma (Lemma 6), we have $\phi_i(u) \geq (3/4)\Phi_i(q)$. Now, consider any throw that occurs at a round $k \geq i$, and suppose this throw targets process q as the victim. We consider two cases. In the first case, the throw is successful with `popTop` returning a node. If the returned node is node u, then after round k, node u has been assigned and possibly already executed. At the very least, node u has been assigned, and the potential has decreased by at least $(2/3)\phi_i(u)$. If the returned node is not node u, then node u has already been assigned and possibly already executed. Again, the potential has decreased by at least $(2/3)\phi_i(u)$. In the other case, the throw is unsuccessful with `popTop` returning `NIL` at either line 4 or line 11. If `popTop` returns `NIL`, then at some time during round k either q's deque was empty or some other `popTop` or `popBottom` returned a topmost node. Either way, by the end of round k, node u has been assigned and possibly executed, so the potential has decreased by at least $(2/3)\phi_i(u)$. In all cases, the potential has decreased by at least $(2/3)\phi_i(u)$. Thus, if a thief targets process q as the victim at a round $k \geq i$, then the potential drops by at least $(2/3)\phi_i(u) \geq (2/3)(3/4)\Phi_i(q) = (1/2)\Phi_i(q)$.

We now consider all P processes and P throws that occur at or after round i. For each process q in D_i, if one or more of the P throws targets q as the victim, then the potential decreases by $(1/2)\Phi_i(q)$. If we think of each throw as a ball toss, then we have an instance of the Balls and Weighted Bins Lemma (Lemma 7). For each process q in D_i, we assign it a weight $W_q = (1/2)\Phi_i(q)$, and for each other process q in A_i, we assign it a weight $W_q = 0$. The weights sum to $W = (1/2)\Phi_i(D_i)$. Using $\beta = 1/2$ in Lemma 7, we conclude that the potential decreases by at least $\beta W = (1/4)\Phi_i(D_i)$ with probability greater than $1 - 1/((1-\beta)e) > 1/4$. ∎

4.3 Analysis for dedicated environments

In this section we analyze the performance of the non-blocking work stealer in dedicated environments. In a dedicated (non-multiprogrammed) environment, all P processes are scheduled in each round, so we have $P_A = P$.

Theorem 9 *Consider any multithreaded computation with work T_1 and critical-path length T_∞ being executed by the non-blocking work stealer with P processes in a dedicated environment. The expected execution time is $O(T_1/P + T_\infty)$. Moreover, for any $\varepsilon > 0$, the execution time is $O(T_1/P + T_\infty + \lg(1/\varepsilon))$ with probability at least $1 - \varepsilon$.*

Proof: Lemma 5 bounds the execution time in terms of the number of throws. We shall prove that the expected number of throws is $O(T_\infty P)$, and that the number of throws is $O((T_\infty + \lg(1/\varepsilon))P)$ with probability at least $1 - \varepsilon$.

We analyze the number of throws by breaking the execution into ***phases*** of $\Theta(P)$ throws. We show that with constant probability, a phase causes the potential to drop by a constant factor, and since we know that the potential starts at $\Phi_0 = 3^{2T_\infty - 1}$ and ends at zero, we can use this fact to analyze the number of phases. The first phase begins at round $t_1 = 1$ and ends at the first round t_1' such that at least P throws occur during the interval of rounds $[t_1, t_1']$. The second phase begins at round $t_2 = t_1' + 1$, and so on.

Consider a phase beginning at round i, and let j be the round at which the next phase begins. We will show that we have $\Pr\{\Phi_j \leq (3/4)\Phi_i\} > 1/4$. Recall that the potential can be partitioned as $\Phi_i = \Phi_i(A_i) + \Phi_i(D_i)$. Since the phase contains at least P throws, Lemma 8 implies that $\Pr\{\Phi_i - \Phi_j \geq (1/4)\Phi_i(D_i)\} > 1/4$. We need to show that the potential also drops by a constant fraction of $\Phi_i(A_i)$. Consider a process q in A_i. If q does not have an assigned node, then $\Phi_i(q) = 0$. If q has an assigned node u, then $\Phi_i(q) = \phi_i(u)$. In this case, process q executes node u at round i and the potential drops by at least $(5/9)\phi_i(u)$. Summing over each process q in A_i, we have $\Phi_i - \Phi_j \geq (5/9)\Phi_i(A_i)$. Thus, no matter how Φ_i is partitioned between $\Phi_i(A_i)$ and $\Phi_i(D_i)$, we have $\Pr\{\Phi_i - \Phi_j \geq (1/4)\Phi_i\} > 1/4$.

We shall say that a phase is ***successful*** if it causes the potential to drop by at least a $1/4$ fraction. A phase is successful with probability at least $1/4$. Since the potential starts at $\Phi_0 = 3^{2T_\infty - 1}$ and ends at 0 (and is always an integer), the number of successful phases is at most $(2T_\infty - 1)\log_{4/3} 3 < 8T_\infty$. The expected number of phases needed to obtain $8T_\infty$ successful phases is at most $32T_\infty$. Thus, the expected number of phases is $O(T_\infty)$, and because each phase contains $O(P)$ throws, the expected number of throws is $O(T_\infty P)$. We now turn to the high probability bound.

Suppose the execution takes $n = 32T_\infty + m$ phases. Each phase succeeds with probability at least $p = 1/4$, so the expected number of successes is at least $np = 8T_\infty + m/4$. We now compute the probability that the number X of successes is less than $8T_\infty$. We use the Chernoff bound [2, Theorem A.13],

$$\Pr\{X < np - a\} < e^{-\frac{a^2}{2np}} ,$$

with $a = m/4$. Thus if we choose $m = 32T_\infty + 16\ln(1/\varepsilon)$, then we have

$$\begin{aligned}
\Pr\{X < 8T_\infty\} &< e^{-\frac{(m/4)^2}{16T_\infty + m/2}} \\
&\leq e^{-\frac{(m/4)^2}{m/2+m/2}} \\
&= e^{-\frac{m}{16}} \\
&\leq e^{-\frac{16\ln(1/\varepsilon)}{16}} \\
&= \varepsilon .
\end{aligned}$$

Thus, the probability that the execution takes $64T_\infty + 16\ln(1/\varepsilon)$ phases or more is less than ε. We conclude that the number of throws is $O((T_\infty + \lg(1/\varepsilon))P)$ with probability at least $1 - \varepsilon$. ■

4.4 Analysis for multiprogrammed environments

We now generalize the analysis of the previous section to bound the execution time of the non-blocking work stealer in multiprogrammed environments. Recall that in a multiprogrammed environment, the kernel is an adversary that may choose not to schedule some of the processes at some or all rounds. In particular, at each round i, the kernel schedules p_i processes of its choosing. We consider three different classes of adversaries, with each class being more powerful than the previous, and we consider increasingly powerful forms of the yield system call. In all cases, we find that the expected execution time is $O(T_1/P_A + T_\infty P/P_A)$.

We prove our upper bounds for multiprogrammed environments using the results of Section 4.2 and the same general approach as is used to prove Theorem 9. The only place in which the proof of Theorem 9 depends on the assumption of a dedicated environment is in the analysis of progress being made by those processes in the set A_i. In particular, in proving Theorem 9, we considered a round i and any process q in A_i, and we showed that at round i, the potential decreases by at least $(5/9)\Phi_i(q)$, because process q executes its assigned node, if any. This conclusion is not valid in a multiprogrammed environment, because the kernel may choose not to schedule process q at round i. For this reason, we need the yield system calls.

The use of yield system calls never constrains the kernel in its choice of the number p_i of processes that it schedules at a step i. Yield calls constrain the kernel only in its choice of *which* p_i processes it schedules. We wish to avoid constraining the kernel in its choice of the number of processes that it schedules, because doing so would admit trivial solutions. For example, if we could force the kernel to schedule only one process, then all we have to do is make efficient use of one processor, and we need not worry about parallel execution or speedup. In general, whenever processors are available and the kernel wishes to schedule our processes on those processors, our user-level scheduler should be prepared to make efficient use of those processors.

4.4.1 Benign adversary

A ***benign*** adversary is able to choose only the number p_i of processes that are scheduled at each round i. It cannot choose which processes are scheduled. The processes are chosen at random. With a benign adversary, the yield system calls are not needed, so line 15 of the scheduling loop (Figure 3) can be removed.

Theorem 10 *Consider any multithreaded computation with work T_1 and critical-path length T_∞ being executed by the non-blocking work stealer with P processes in a multiprogrammed environment. In addition, suppose the kernel is a benign adversary, and the yield system call does nothing. The expected execution time is $O(T_1/P_A + T_\infty P/P_A)$. Moreover, for any $\varepsilon > 0$, the execution time is $O(T_1/P_A + (T_\infty + \lg(1/\varepsilon))P/P_A)$ with probability at least $1 - \varepsilon$.*

Proof: As in the proof of Theorem 9, we bound the number of throws by showing that in each phase, the potential decreases by a constant factor with constant probability. We consider a phase that begins at round i. The potential is $\Phi_i = \Phi_i(A_i) + \Phi_i(D_i)$. From Lemma 8, we know that the potential decreases by at least $(1/4)\Phi_i(D_i)$ with probability more than $1/4$. It remains to prove that with constant probability the potential also decreases by a constant fraction of $\Phi_i(A_i)$.

Consider a process q in A_i. If q is scheduled at some round during the phase, then the potential decreases by at least $(5/9)\Phi_i(q)$ as in Theorem 9. During the phase, at least P throws occur, so at least P processes are scheduled, with some processes possibly being scheduled multiple times. These scheduled processes are chosen at random, so we can treat them like random ball tosses and appeal to the Balls and Weighted Bins Lemma (Lemma 7). In fact, this selection of processes at random does not correspond to independent ball tosses, because a process cannot be scheduled more than once in a given round, which introduces dependencies. But these dependencies only increases the probability that a bin receives a ball. (Here each deque is a bin and a bin is said to receive a ball if and only if the associated process is scheduled.) We assign each process q in A_i a weight $W_q = (5/9)\Phi_i(q)$ and each process q in D_i a weight $W_q = 0$. The total weight is $W = (5/9)\Phi_i(A_i)$, so using $\beta = 1/2$ in Lemma 7, we conclude that the potential decreases by at least $\beta W = (5/18)\Phi_i(A_i)$ with probability greater than $1/4$.

The event that the potential decreases by $(5/18)\Phi_i(A_i)$ is independent of the event that the potential decreases by $(1/4)\Phi_i(D_i)$, because the random choices of which processes to schedule are independent of the random choices of victims. Thus, both events occur with probability greater than $1/16$, and we conclude that the potential decreases by at least $(1/4)\Phi_i$ with probability greater than $1/16$. The remainder of the proof is the same as that of Theorem 9, but with different constants. ■

4.4.2 Oblivious adversary

An ***oblivious*** adversary is able to choose both the number p_i of processes and which p_i processes are scheduled at each round i, but is required to make these decisions in an off-line manner. Specifically, before the execution begins the oblivious adversary commits itself to a complete kernel schedule.

To deal with an oblivious adversary, we employ a directed yield [1, 28] to a random process; we call this operation `yieldToRandom`. If at round i process q calls `yieldToRandom`, then a random process r is chosen and the kernel cannot schedule process q again until it has scheduled process r. More precisely, the kernel cannot schedule process q at a round $j > i$ unless there exists a round k, $i \le k \le j$, such that process r is scheduled at round k. Of course, this requirement may be inconsistent with the kernel schedule. Suppose process q is scheduled at rounds i and j, and process r is not scheduled at any round $k = i, \ldots, j$. In this case, if q calls `yieldToRandom` at round i, then because q cannot be scheduled at round j as the schedule calls for, we schedule process r instead. That is, we schedule process r in place of q. Observe that this change in the schedule does not change the number of processes scheduled at any round; it only changes which processes are scheduled.

The non-blocking work stealer uses `yieldToRandom`. Specifically, line 15 of the scheduling loop (Figure 3) is `yieldToRandom()`.

Theorem 11 *Consider any multithreaded computation with work T_1 and critical-path length T_∞ being executed by the non-blocking work stealer with P processes in a multiprogrammed environment. In addition, suppose that the kernel is an oblivious adversary, and the yield system call is* `yieldToRandom`*. The expected execution time is $O(T_1/P_A + T_\infty P/P_A)$. Moreover, for any $\varepsilon > 0$, the execution time is $O(T_1/P_A + (T_\infty + \lg(1/\varepsilon))P/P_A)$ with probability at least $1 - \varepsilon$.*

Proof: As in the proof of Theorem 10, it remains to prove that in each phase, the potential decreases by a constant fraction of $\Phi_i(A_i)$ with constant probability. Again, if q in A_i is scheduled at a round during the phase, then the potential decreases by at least $(5/9)\Phi_i(q)$. Thus, if we can show that in each phase at least P processes chosen at random are scheduled, then we can appeal to the Balls and Weighted Bins Lemma.

Whereas previously we defined a phase to contain at least P throws, we now define a phase to contain at least $2P$ throws. With at least $2P$ throws, at least P of these throws have the following property: The throw was performed by a process q at a round j during the phase, and process q also performed another throw at a round $k > j$, also during the phase. We say that such a throw is ***followed***. Observe that in this case, process q called `yieldToRandom` at some round between rounds j and k. Since process q is scheduled at round k, the victim process is scheduled at some round between j and k. Thus, for every throw that is followed, there is a randomly chosen victim process that is scheduled during the phase.

Consider a phase that starts at round i, and partition the steal attempts into two sets, F and G, such that every throw in F is followed, and each set contains at least P throws. Because the phase contains at least $2P$ throws and at least P of them are followed, such a partition is possible. Lemma 8 tells us that the throws in G cause the potential to decrease by at least $(1/4)\Phi_i(D_i)$ with probability greater than $1/4$. It remains to prove that the throws in F cause the potential to decrease by a constant fraction of $\Phi_i(A_i)$.

The throws in F give rise to at least P randomly chosen victim processes, each of which is scheduled during the phase. Thus, we treat these P random choices as ball tosses, assigning each process q in A_i a weight $W_q = (5/9)\Phi_i(q)$, and each other process q in D_i a weight $W_q = 0$. We then appeal to the Balls and Weighted Bins Lemma with $\beta = 1/2$ to conclude that the throws in F cause the potential to decrease by at least $\beta W = (5/18)\Phi_i(A_i)$ with probability greater than $1/4$. Note that if the adversary is not oblivious, then we cannot treat these randomly chosen victim processes as ball tosses, because the adversary can bias the choices away from processes in A_i. In particular, upon seeing a throw by process q target a process in A_i as the victim, an adaptive adversary may stop scheduling process q. In this case the throw will not be followed, and hence, will not be in the set F. The oblivious adversary has no such power.

The victims targeted by throws in F are independent of the victims targeted by throws in G, so we conclude that the potential decreases by at least $(1/4)\Phi_i$ with probability greater than $1/16$. The remainder of the proof is the same as that of Theorem 9, but with different constants. ∎

4.4.3 Adaptive adversary

An ***adaptive*** adversary selects both the number p_i of processes and which of the p_i processes execute at each round i, and it may do so in an on-line fashion. The adaptive adversary is constrained only by the requirement to obey yield system calls.

To deal with an adaptive adversary, we employ a powerful yield that we call `yieldToAll`. If at round i process q calls `yieldToAll`, then the kernel cannot schedule process q again until it has scheduled every other process. More precisely, the kernel cannot schedule process q at a round $j > i$, unless for every other process r, there exists a round k_r in the range $i \le k_r \le j$, such that process r is scheduled at round k_r. Note that `yieldToAll` does not constrain the adversary in its choice of the number of processes scheduled at any round. It constrains the adversary only in its choice of which processes it schedules.

The non-blocking work stealer calls `yieldToAll` before each steal attempt. Specifically, line 15 of the scheduling loop (Figure 3) is `yieldToAll()`.

Theorem 12 *Consider any multithreaded computation with work T_1 and critical-path length T_∞ being executed by the non-blocking work stealer with P processes in a multiprogrammed environment. In addition, suppose the kernel is an adaptive adversary, and the yield system call is* `yieldToAll`. *The expected execution time is $O(T_1/P_A + T_\infty P/P_A)$. Moreover, for any $\varepsilon > 0$, the execution time is $O(T_1/P_A + (T_\infty + \lg(1/\varepsilon))P/P_A)$ with probability at least $1 - \varepsilon$.*

Proof: As in the proofs of Theorems 10 and 11, it remains to argue that in each phase the potential decreases by a constant fraction of $\Phi_i(A_i)$ with constant probability. We define a phase to contain at least

$2P+1$ throws. Consider a phase beginning at round i. Some process q executed at least three throws during the phase, so it called `yieldToAll` at some round before the third throw. Since q is scheduled at some round after its call to `yieldToAll`, every process is scheduled at least once during the phase. Thus, the potential decreases by at least $(5/9)\Phi_i(A_i)$. The remainder of the proof is the same as that of Theorem 9. ∎

5 Related work

Prior work on thread scheduling has not considered multiprogrammed environments, but in addition to proving time bounds, some of this work has considered bounds on other metrics of interest, such as space and communication. For the restricted class of "fully strict" multithreaded computations, the work stealing algorithm is efficient with respect to both space and communication [8]. Moreover, when coupled with "dag-consistent" distributed shared memory, work stealing is also efficient with respect to page faults [6]. For these reasons, work stealing is practical and variants have been implemented in many systems [7, 19, 20, 24, 34, 38]. For general multithreaded computations, other scheduling algorithms have also been shown to be simultaneously efficient with respect to time and space [4, 5, 13, 14]. Of particular interest here is the idea of deriving parallel depth-first schedules from serial schedules [4, 5], which produces strong upper bounds on time and space. The practical application and possible adaptation of this idea to multiprogrammed environments is an open question.

Prior work that has considered multiprogrammed environments has focused on the kernel-level scheduler. With coscheduling (also called gang scheduling) [18, 33], all of the processes belonging to a computation are scheduled simultaneously, thereby giving the computation the illusion of running on a dedicated machine. Interestingly, it has recently been shown that in networks of workstations coscheduling can be achieved with little or no modification to existing multiprocessor operating systems [17, 35]. Unfortunately, for some job mixes, coscheduling is not appropriate. For example, a job mix consisting of one parallel computation and one serial computation cannot be coscheduled efficiently. With process control [36], processors are dynamically partitioned among the running computations so that each computation runs on a set of processors that grows and shrinks over time, and each computation creates and kills processes so that the number of processes matches the number of processors. We are not aware of any commercial operating system that supports process control.

6 Conclusion

Whereas traditional thread schedulers demonstrate poor performance in multiprogrammed environments [9, 15, 17, 23], the non-blocking work stealer executes with guaranteed high performance in such environments. By implementing the work-stealing algorithm with non-blocking deques and judicious use of yield system calls, the non-blocking work stealer executes any multithreaded computation with work T_1 and critical-path length T_∞, using any number P of processes, in expected time $O(T_1/P_A + T_\infty P/P_A)$, where P_A is the average number of processors on which the computation executes. Thus, it achieves linear speedup — that is, execution time $O(T_1/P_A)$ — whenever the number of processes is small relative to the parallelism T_1/T_∞ of the computation. Moreover, this bound holds even when the number of processes exceeds the number of processors and even when the computation runs on a set of processors that grows and shrinks over time. We prove this result under the assumption that the kernel, which schedules processes on processors and determines P_A, is an adversary.

We have implemented the non-blocking work stealer in a prototype `C++` threads library called ***Hood*** [10]. For UNIX platforms, Hood is built on top of POSIX threads [29] that provide the abstraction of processes (known as "system-scope threads" or "bound threads"). For performance, the deque methods are coded in assembly language. For the yields, Hood employs a combination of the UNIX `priocntl` (priority control) and `yield` system calls to implement a `yieldToAll`. Using Hood, we have coded up several applications, and we have run numerous experiments, the results of which attest to the practical application of the non-blocking work stealer. These empirical results [9, 10] show that application performance does conform to our analytical bound and that the constant hidden inside the big-Oh notation is small — roughly 1.

Acknowledgments

Coming up with a correct non-blocking implementation of the deque data structure was not easy, and we have several people to thank. Keith Randall of MIT found a bug in an early version of our implementation, and Mark Moir of The University of Pittsburgh suggested ideas that lead us to a correct implementation. Keith also gave us valuable feedback on a draft of this paper. We thank Dionisios Papadopoulos of UT Austin, who has been collaborating on our implementation and empirical study of the non-blocking work stealer. Finally, we thank Charles Leiserson and Matteo Frigo of MIT and Geeta Tarachandani of UT Austin for listening patiently as we tried to hash out some of our early ideas.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, July 1986.

[2] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. John Wiley & Sons, 1992.

[3] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[4] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, California, July 1995.

[5] 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*, pages 12–23, Newport, Rhode Island, June 1997.

[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 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[7] 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.

[8] 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*, pages 356–368, Santa Fe, New Mexico, November 1994.

[9] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments (extended abstract). In *Proceedings of the 1998 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Poster Session*, Madison, Wisconsin, June 1998.

[10] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. `http://www.cs.utexas.edu/users/hood`, 1999.

[11] Robert D. Blumofe, C. Greg Plaxton, and Sandip Ray. Verification of a concurrent deque implementation. Technical Report TR–99–11, Department of Computer Science, University of Texas at Austin, June 1999.

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

[13] F. Warren Burton. Guaranteeing good space bounds for parallel programs. Technical Report 92-10, Simon Fraser University, School of Computing Science, November 1992.

[14] F. Warren Burton and David J. Simpson. Space efficient execution of deterministic parallel programs. Unpublished manuscript, 1994.

[15] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, December 1991.

[16] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, England, 1968. Originally published as Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[17] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, Philadelphia, Pennsylvania, May 1996.

[18] Dror G. Feitelson and Larry Rudolph. Coscheduling based on runtime identification of activity working sets. *International Journal of Parallel Programming*, 23(2):135–160, April 1995.

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

[20] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

[21] 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*, pages 212–223, Montreal, Canada, June 1998.

[22] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[23] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, San Diego, California, May 1991.

[24] 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.

[25] M. Herlihy and J. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[26] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Seattle, Washington, March 1990.

[27] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[28] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, Saint-Malo, France, October 1997.

[29] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads*. SunSoft Press, Prentice Hall, 1996.

[30] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, San Diego, California, June 1992.

[31] 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.

[32] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 219–228, Santa Barbara, California, August 1997.

[33] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[34] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, July 1994.

[35] Patrick G. Sobalvarro and William E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 106–126, April 1995.

[36] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159–166, Litchfield Park, Arizona, December 1989.

[37] Jeffrey D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

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