Helper Locks for Fork-Join Parallel Programming

Kunal Agrawal* Charles E. Leiserson Jim Sukha

MIT Computer Science and Artificial Intelligence Laboratory

kunal@cse.wustl.edu cel@mit.edu sukhaj@mit.edu

Abstract

Helper locks allow programs with large parallel critical sections, called parallel regions, to execute more efficiently by enlisting processors that might otherwise be waiting on the helper lock to aid in the execution of the parallel region. Suppose that a processor p is executing a parallel region A after having acquired the lock L protecting A. If another processor p' tries to acquire L, then instead of blocking and waiting for p to complete A, processor p' joins p to help it complete A. Additional processors not blocked on L may also help to execute A.

The HELPER runtime system can execute fork-join computations augmented with helper locks and parallel regions. HELPER supports the unbounded nesting of parallel regions. We provide theoretical completion-time and space-usage bounds for a design of HELPER based on work stealing. Specifically, let V be the number of parallel regions in a computation, let T_1 be its work, and let $\widetilde{T}_\infty$ be its "aggregate span" — the sum of the spans (critical-path lengths) of all its parallel regions. We prove that HELPER completes the computation in expected time $O(T_1/P+\widetilde{T}_\infty+PV)$ on P processors. This bound indicates that programs with a small number of highly parallel critical sections can attain linear speedup. For the space bound, we prove that HELPER completes a program using only $O(P\widetilde{S}_1)$ stack space, where $\widetilde{S}_1$ is the sum, over all regions, of the stack space used by each region in a serial execution. Finally, we describe a prototype of HELPER implemented by modifying the Cilk multithreaded runtime system. We used this prototype to implement a concurrent hash table with a resize operation protected by a helper lock.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms, Performance, Theory

Keywords Cilk, fork-join multithreading, helper lock, nested parallelism, parallel region, scheduling, work stealing.

*Kunal Agrawal's current affiliation is Washington University in St. Louis.

This research was supported in part by NSF Grants CNS-0540248 and CNS-0615215.

PPoPP'10, January 9–14, 2010, Bangalore, India.

1. INTRODUCTION

Many multithreaded-programming environments such as Cilk [6], Cilk++ [9], Fortress [1], Threading Building Blocks [12], and X10 [5] contain constructs for supporting fork-join parallelism. In this paradigm, programmers specify parallelism in a program abstractly by permitting (but not requiring) parallelism in specific locations of the program. A runtime system dynamically schedules the program on P processors, called ***workers***.

Environments that support fork-join programming usually support ***nested parallelism***. For example, a programmer can "spawn" two nested tasks `G1` and `G2` from inside a task `F`, indicating that `G1` and `G2` can be executed in parallel. The tasks `G1` and `G2` can spawn more subtasks, and so on. The runtime does not, however, create a new thread every time a new nested task is spawned. Depending on how many workers are available, the runtime may choose to execute `G1` and `G2` serially or in parallel.

Some environments that support fork-join programming preclude non-fork-join constructs in order to exploit efficient scheduling algorithms that exist for programs that have only fork-join dependencies. For example, some multithreaded systems use a scheduler modeled after the work-stealing scheduler implemented in Cilk [6], which is known to be provably efficient in terms of both completion time and stack space [4].

Unfortunately, the theoretical bounds in [4] do not directly apply to programs that use locks, because locks can introduce arbitrary dependencies. The completion-time bound in [4] assumes that two concurrently spawned tasks `G1` and `G2` can execute independently. If `G1` and `G2` attempt to acquire the same lock, however, this assumption may not hold, since one task must wait for the other. If the lock protects large critical sections in `G1` or `G2`, the lock may compromise scalability. Alternatively, if one modifies the scheduler so that workers randomly work-steal instead of waiting when they block trying to acquire a lock, then, as we shall show in Section 2, the program can consume exorbitant stack space, since workers may repeatedly fail to acquire locks.

Helper locks address these issues. Intuitively, a helper lock is like an ordinary lock, but it is "connected" to a large critical section containing nested parallelism, called a ***parallel region***. Whenever a worker fails to acquire a helper lock L, it tries to help complete the work in the parallel region A connected to L. Helper locks can improve program performance, because if a worker blocks trying to acquire L, it can help complete useful work in A, rather than just waiting. Also, because helper locks direct a worker toward specific work, the stack space used by a program can be controlled.

This paper makes the following contributions:

- The design of the HELPER runtime system, which can execute fork-join computations augmented with helper locks and parallel regions of unbounded nesting depth.
- Theoretical bounds on the completion time and stack-space usage of computations executed using HELPER.

- A prototype of HELPER based on the Cilk runtime system, which suggests that helper locks can be implemented with reasonable overheads.

Our theoretical work extends the results given in [2, 4] for work-stealing schedulers, showing that for a computation $\mathcal{E}$, HELPER completes $\mathcal{E}$ on P processors in expected time $O(T_1/P + \widetilde{T}_\infty + PV)$, where T_1 is the work of $\mathcal{E}$, $\widetilde{T}_\infty$ is $\mathcal{E}$'s "aggregate span" which is bounded by the sum of spans (critical-path lengths) of all regions, and V is the number of parallel regions in $\mathcal{E}$. Our completion-time bounds are asymptotically optimal for certain computations with parallel regions and helper locks. In addition, the bounds imply that HELPER produces linear speedup provided that all parallel regions in the computation are sufficiently parallel. Roughly, if for every region A, the nonnested work of region A is asymptotically larger than P times the span of A, then HELPER executes the computation with speedup approaching P. We also show that HELPER completes $\mathcal{E}$ using only $O(P\widetilde{\mathcal{S}}_1)$ stack space, where $\widetilde{\mathcal{S}}_1$ is the sum over all regions A of the stack space used by A in a serial execution of the same computation $\mathcal{E}$.

As a proof-of-concept, we implemented a prototype of HELPER by modifying the Cilk [6] runtime system. We used the HELPER prototype to program a concurrent hash table that uses helper locks to protect resize operations. In this hash table, a resize rebuilds the entire table, and thus it must hold a lock to prevent inserts from interfering. By protecting the resize with a helper lock, workers that fail to insert can help to complete the resize instead of waiting. We performed experiments which suggest that a practical and efficient implementation of HELPER is feasible.

Outline

The rest of this paper is organized as follows. Section 2 explores the example of a concurrent resizable hash table and the challenges posed by large critical sections, explaining how helper locks can address these challenges. Section 3 presents the design of HELPER, focusing on the runtime support for helper locks. Section 4 states the theoretical bounds on completion time and space usage for HELPER. Sections 5 and 6 give details of the proof of these bounds. Section 7 describes the prototype implementation of HELPER, and Section 8 presents experimental results on our prototype system for a simple concurrent hash-table benchmark. Section 9 concludes with related work and future research directions.

2. MOTIVATING EXAMPLE

This section motivates the utility of helper locks through the example of coding a resizable hash table written using Cilk [6], an open-source fork-join multithreaded programming language. First, we briefly review key features of the Cilk language and runtime. Then, we discuss the challenges of using Cilk with ordinary locks to exploit parallelism within a hash table's resize operation. Finally, we explain helper locks and how they can be used to run the hash-table example more simply and efficiently.

Overview of Cilk

We review the characteristics of Cilk and its work-stealing scheduler using the sample program in Figure 1. This pseudocode shows a Cilk function that concurrently inserts n random keys into a resizable hash table.

Cilk extends C with two main keywords: `spawn` and `sync`. In Cilk, the `spawn` keyword before a function invocation specifies that this ***child*** function can potentially execute in parallel with the ***continuation*** of the ***parent*** (caller), that is, the code that immediately follows the spawn statement in the parent. The `sync` keyword precludes any code after the `sync` statement from executing until all previously spawned children of the parent have completed. As an example, in Figure 1, the `rand_inserts` function uses the `spawn` and `sync` keywords to perform n insert operations in parallel in a divide-and-conquer fashion.

```
cilk void rand_inserts(HashTable* H, int n) {
  if (n <= 32) { random_inserts_serial(H, n); }
  else {
    spawn rand_inserts(H, n/2);
    spawn rand_inserts(H, n-n/2);
    sync;
  }
}
void rand_inserts_serial(HashTable* H, int n) {
  for (int i = 0; i < n; i++) {
    int res; Key k = rand();
    do {
      res = try_insert(H, k, k);
    } while (res == FAILED);
    resize_table_if_overflow(H);
  }
}
```

Figure 1. An example Cilk function which performs n hash table insertions, potentially in parallel. After every insertion, the `rand_inserts` method checks whether the insertion triggered an overflow and resizes the table if necessary.

The Cilk runtime executes a program on P workers, where P is determined at runtime. Conceptually, every worker stores its work on a double-ended queue, or ***deque***. When a function f being executed by a worker spawns a function f', the worker suspends f, pushes the continuation of f onto the bottom (tail) of its deque, and begins working on f'. When the worker completes f', it pops f from the bottom of its deque and resumes it at the point of the continuation. When a worker's deque becomes empty or the executing function stalls at a `sync`, however, the worker chooses a victim worker uniformly at random and tries to ***steal*** work from the top (head) of the victim's deque.

Challenges for Large Critical Sections

Cilk programmers can use ordinary locks to protect critical sections. For example, the code in Figure 2 uses a reader/writer lock to implement a resizable concurrent hash table. Every insert operation acquires the table's reader lock, and a resize operation acquires the table's writer lock. Thus, insert operations (on different buckets) may run in parallel, but a table resize cannot execute in parallel with any insertion.

When a worker blocks trying to acquire a lock, the worker typically spins until the lock is released. Workers that block on a lock protecting a large critical section may waste a substantial number of processor cycles waiting for its release. Consequently, most lock implementations avoid tying up the blocked worker thread by yielding the scheduling quantum after spinning for a short length of time. This altruistic strategy works well if there are other jobs that can use the cycles in a multiprogrammed environment, but if the focus is on completing the computation, it seems better to put the blocked workers to work on the computation itself.

A naive strategy for putting a blocked worker to work on the computation itself is for the worker to suspend the function that failed to acquire a lock and engage in work-stealing. Unfortunately, this strategy can waste resources in dramatic fashion. Turning back to the example of the hash table, imagine what would happen if one worker p write-acquires the resize lock while another worker p' attempts an insertion. Unable to read-acquire the resize lock, p' would suspend the insertion attempt and steal work. What work is lying around and available to steal? Why another insertion, of course! Indeed, while p is tooling away trying to resize, p' might

```
1  int try_insert(HashTable* H, Key k, void* value) {
2    int success = 0;
3    success = try_read_acquire(H->resize_lock);
4    if (!success) { return FAILED; }
5    int idx = hashcode(H, k);
6    List* L = H->buckets[idx];
7    list_lock(L);
8    list_insert(L, k, value);
9    list_unlock(L);
10   release(H->resize_lock);
11   return SUCCESS;
12 }

13 void resize_table_if_overflow(HashTable* H) {
14   if (is_overflow(H)) {
15     write_acquire(H->resize_lock);
16     List** new_buckets;
17     int new_n = H->num_buckets*2;
18     new_buckets = create_buckets(new_size);
19     for (int i = 0; i < H->num_buckets; i++) {
20       rehash_list(H->buckets[i], new_buckets, new_n);
21     }
22     free_buckets(H->buckets);
23     H->buckets = new_buckets;
24     H->num_buckets = new_n;
25     release(H->resize_lock);
26   }
27 }
```

Figure 2. Code for inserting into and resizing a concurrent hash table using a reader/writer lock.

attempt (and fail) to insert most of the items in the hash table, one at a time, systematically suspending each insertion as it fails to read-acquire the resize lock and going on to the next.

This strategy is not only wasteful of p''s efforts, it results in profligate space usage. Each time a continuation is stolen, the runtime system requires an activation record to store local variables. Thus, for the hash table example, the `rand_inserts` function could generate as much as $\Theta(n)$ space in suspended insertions. In general, the space requirement could grow as large as the total work in the computation. In contrast, Cilk's strategy of simply spinning and yielding, though potentially wasteful of processor cycles, uses space of at most $\Theta(P \lg n)$ on P processors in this example.

The idea of helper locks is to employ the blocked workers in productive work while controlling space usage by enlisting them to help complete a large critical section. For helper locks to be useful, however, the critical section must be parallelized and the runtime system must be able to migrate blocked workers to work on the critical section. Ordinary Cilk-style work-stealing is inadequate to this task, because stealing occurs at the top of the deque, and the critical section may be deeply nested where workers cannot find it. For example, in Figure 2, when an insertion triggers a resize, the parallel work of the resize (i.e., `rehash_list`) is generated at the bottom of a deque, with work for `rand_inserts` above it. In fact, in this example, by employing Cilk-style work-stealing, workers would more likely block on another insertion than help complete the resize.

Helper Locks

Helper locks and the runtime support provided by HELPER allow blocked workers to productively help to complete parallel critical sections, or ***parallel regions***. To specify a parallel region, the programmer encapsulates the critical section in a function and precedes the invocation of the function with the `start_region` keyword. A ***helper lock*** extends an ordinary lock by providing a method `helper_acquire` for acquiring the lock. (For a

```
1 CilkHelperLock* L = H->resize_lock;
2 helper_read_acquire(L);
3 try_insert(H, k, k);
4 helper_release(L);
5 if (is_overflow(H)) {
6   helper_write_acquire(L);
7   start_region resize_if_overflow(H);
8 }
```

Figure 3. Pseudocode for a resizable hash table using a helper lock `L`. This code represents a modification of the inner loop of the `rand_inserts` function (lines 12–15 of Figure 1).

reader/writer lock, two methods `helper_read_acquire` and `helper_write_acquire` would be provided). When a worker fails to acquire a helper lock L, it helps complete the work of a designated parallel region A ***connected to*** the lock L.

To be precise, a worker p succeeds in a call `helper_acquire`(L) exactly when an ordinary acquire of L would succeed, and it provides the same exclusion guarantees as the original lock. First, we describe the case where p succeeds in a helper-acquire of L. There are two ways to execute L's critical section and release L. If L protects a serial critical section, then the programmer can just execute the critical section and call `helper_release`(L), in which case, L behaves like its underlying ordinary lock. A lock acquisition ending with a helper-release is called a ***short acquire***. If, on the other hand, the programmer calls `start_region` on a parallel region A while holding a helper lock L, then L protects A. (Calling `start_region` while holding an ordinary lock is disallowed.) Conceptually, a `start_region` call transfers ownership of any locks L that p has acquired but not released to region A, and connects any such L to region A. After region A completes, it automatically releases all helper locks that it holds. An acquisition that ends with the return from a region is called a ***region acquire***. Nested `start_region` calls behave similarly, except that an inner call transfers and releases ownership of only those helper locks acquired at the current nesting level. In the case of short acquires, the processor that acquires the locks conceptually owns the locks and is responsible for releasing them. In the case of region acquires, the parallel region owns the locks and the locks are automatically released (by the runtime system) when the region completes.

A worker p_1 attempting to acquire a helper lock L can fail either because some parallel region A holds L or because some other worker p_2 holds L. If a region A holds L, then p_1 tries to help complete the parallel work in A, since L is connected to A. If p_2 holds L, then p_1 waits (or waits and yields the scheduling quantum) as with an ordinary failed lock acquisition.

Helper locks help programmers deal with large critical sections in two ways. When a worker p blocks on a helper lock L, it keeps productively busy by helping to complete the region A holding L (if such a region exists). Moreover, it enables faster release of the lock L on which p is blocked. In addition, this strategy guarantees (see Section 6) that the stack space can be bounded.

Figure 3 illustrates how one might modify the code with a reader/writer lock in Figure 2 to use a reader/writer helper lock L. In Lines 2–4, the hash table insert is protected by a helper-acquire of L in read mode. Line 6 performs a helper-acquire of L in write mode, and then Line 7 starts the region for the resize. With a helper lock, one can potentially execute the resize in parallel, e.g., by spawning the `rehash_list` calls in Line 20 of Figure 2. If any worker p fails to acquire L in either read or write mode because a resize region A holds L, then p tries to help complete the resize.

HELPER requires that parallel regions be properly nested, as is enforced by the linguistics of `start_region`. Short acquires need not obey this restriction and can be used with other locking disciplines, such as hand-over-hand locking. HELPER does assume that locks are not reentrant, however. Our prototype implementation also assumes that A region acquire does not operate on reader helper locks, but only on mutex (and writer) helper locks, although a short acquire can operate on any kind of helper lock. This restriction is not inherent in the HELPER design, but it simplifies analysis and implementation.

3. THE HELPER RUNTIME SYSTEM

This section describes the HELPER runtime system, which provides support for helper locks using a Cilk-like work-stealing scheduler. HELPER provides two new runtime mechanisms: the `start_region` mechanism which creates a new parallel region, and a `help_region` mechanism which helper locks use to assign blocked workers to parallel regions. We present HELPER in the context of Cilk [6], the system we used to implement our prototype. The design can be applied more generally to other fork-join parallel languages that use a work-stealing scheduler.

Parallel Regions

A parallel region is an execution instance of a critical section created by `start_region` and is protected by one or more helper locks. When a worker blocks due to a helper-lock acquire, it tries to help in a parallel region that holds that lock (if such a parallel region exists). To manage worker migration, the runtime maintains a pool of deques for each active (currently executing) parallel region.

When the programmer calls the `start_region` method to create a parallel region A, the runtime system creates a new ***deque pool*** for A, denoted by $\mathtt{dqpool}(A)$. While A is being executed, A has certain workers ***assigned*** to it. The runtime system allocates a deque $q \in \mathtt{dqpool}(A)$ to every worker p when p is assigned to A. We denote this relationship by $A = \mathtt{region}(q)$ or $\mathtt{dq}(p,A) = q$. We say $\mathtt{dq}(p,A) = \mathtt{NULL}$ if p is not assigned to A. The runtime uses $\mathtt{dqpool}(A)$ for self-contained scheduling of region A on A's assigned workers. While p is assigned to A, when p tries to steal work, it randomly steals only from deques $q \in \mathtt{dqpool}(A)$.

HELPER allows a worker p to enter (be assigned to) a region A in three ways. First, p can be assigned to A when p successfully starts region A by executing `start_region`. Second, p can enter if it blocks on a lock L held by region A. In this case, the runtime system executes a `help_region` call on behalf of the worker, and the worker is assigned to A. Finally, p can enter a region A due to random work stealing: if p tries to steal from p' and discovers that p' is assigned to A, then p may also enter A.

Conceptually, a worker may leave a parallel region before the region completes, but early leaving raises several issues. For example, a worker might repeatedly leave and enter the same region, repeatedly incurring the synchronization overhead. Therefore, for simplicity in implementation and to guarantee good theoretical bounds, once HELPER, assigns a worker to a region, it remains in that until the region is done.

We maintain a deque pool as an array of size P with a dedicated slot for every worker. Every worker adds or removes themselves from the pool without waiting on other workers. A protocol based on a binary tree network of size $O(P)$ can track whether the pool is empty. With this scheme, each worker waits at most $O(\lg P)$ time to enter the region, and it takes at most $O(\lg P)$ time for all workers to leave the deque pool once all work in the region is completed.

Nested Regions

HELPER supports nested helper locks and nested parallel regions. That is, a region A protected by region lock L_1 can start a region B protected by lock L_2 or call `help_region` on L_2. With nested locks, every worker p may be assigned to many parallel regions

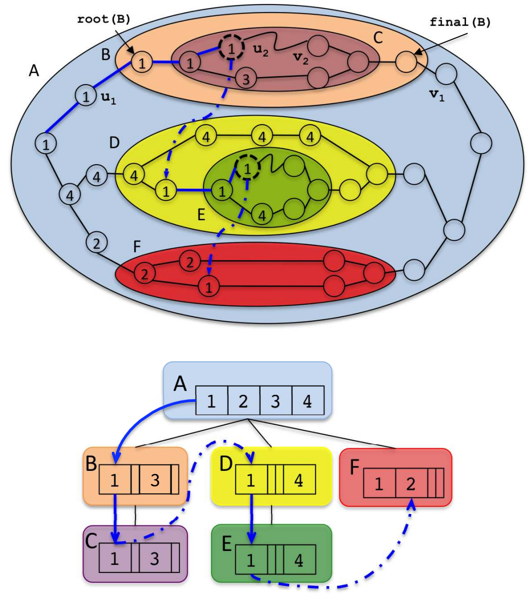

Figure 4. (a) A computation dag showing parallel regions. Ovals represent regions, and nodes are labeled with the number of the worker that executes each node (e.g., worker p_1 executes the nodes labeled 1). The dotted nodes correspond to `help_region` calls. (b) A snapshot of deque pools during execution, where numbers correspond to workers in the pool.

and thus have many deques. HELPER supports nesting of arbitrary depth by maintaining a chain of deques for each worker.

The deques for one worker form a ***deque chain*** with each deque along the chain belonging to the deque pool of a distinct region. The top deque in every worker's chain belongs to the global deque pool, which is the original set of deques for a normal Cilk program context. The bottom deque in p's chain represents p's ***active deque***, denoted $\mathtt{activeDQ}(p)$. When p is working normally, it changes only the tail of $\mathtt{activeDQ}(p)$. In addition, p always work-steals from deques within the deque pool of $\mathtt{region}(\mathtt{activeDQ}(p))$, the region for p's active deque.

Whenever a worker p with $\mathtt{activeDQ}(p) = \mathtt{dq}(p,A)$ enters a region B, it adds a new deque $\mathtt{dq}(p,B)$ for region B to the bottom of its chain and changes $\mathtt{activeDQ}(p)$ to $\mathtt{dq}(p,B)$. We say that $\mathtt{dq}(p,B)$ has a ***child deque*** of $\mathtt{dq}(p,A)$, i.e., $\mathtt{child}(\mathtt{dq}(p,A)) = \mathtt{dq}(p,B)$, and we define ***parent deque*** similarly, i.e., $\mathtt{parent}(\mathtt{dq}(p,B)) = \mathtt{dq}(p,A)$.

When B completes, for every worker p assigned to A, p removes its deque $q_p = \mathtt{dq}(p,B)$ from the end of its deque chain, sets the parent of q_p as its active deque, and starts working on region $\mathtt{region}(\mathtt{parent}(q_p))$. Note that different workers assigned to B may return to different regions.

Deque chains also help a worker p efficiently find deque pools for other regions during random work-stealing. A worker p_1 with an empty active deque randomly steals from other pools in the same region $A = \mathtt{region}(\mathtt{activeDQ}(p_1))$. If p_1 finds a deque $q = \mathtt{dq}(p_2,A)$ which is also empty, but a child deque $q' = \mathtt{child}(q)$ exists, then instead of failing the steal attempt, p_1 enters the region corresponding to q'. If q' is also empty but $\mathtt{child}(q')$ exists, etc., p_1 can continue to enter the regions for deques deeper in the chain.

Figure 4 illustrates deque pools and deque chains for a simple computation using 4 workers. (The dag model for a computation is presented in Section 4.) In this example, worker p_1 enters regions

A through F (all regions are assumed to acquire different helper locks). Initially, p_1 starts a region B, p_4 randomly steals from p_1 in $\texttt{dqpool}(A)$, and starts region D, and p_2 steals from p_4 in A, and starts region F. Next, p_1 starts a region C nested inside B. Then, p_3 randomly work-steals from p_1 in A, and enters B and C. Afterward, p_1 inside C makes a `help_region` call on the lock for D, enters D, and steals from p_4 in D. Finally, p_1 makes a `help_region` call on the lock for F and enters F.

Deadlock Freedom

As with ordinary (nonhelper) locks, an arbitrary nesting of helper locks risks deadlock. If program with ordinary locks protecting serial critical sections is deadlock free, however, converting the ordinary locks into helper locks is safe and does not introduce any new deadlocks. (The discipline often used to ensure deadlock freedom is to acquire locks in a fixed order.) After the conversion to helper locks, large serial critical sections can be parallelized, but the programmer must ensure that any nested parallelism within critical sections is properly encapsulated using parallel regions. Specifically, the keywords `spawn` and `sync` should not appear inside a critical section unless both are enclosed within a parallel region initiated using `start_region`. This condition essentially requires that programmers use only the mechanisms provided by HELPER to generate nested parallelism in critical sections.

4. COMPLETION TIME AND SPACE USAGE

This section states the completion time and space bounds provided by HELPER and provides an interpretation. The proofs are presented in Sections 5 and 6. We begin with some definitions.

Definitions

We model the execution of a program as a computation dag $\mathcal{E}$. Each node in $\mathcal{E}$ represents a unit-time task, and each edge represents a dependence between tasks. We assume the computation executes on hardware with P processors, with one ***worker*** assigned to each processor. In the remainder of this paper, we consider only computations $\mathcal{E}$ generated by deadlock-free programs.

We model regions as subdags of $\mathcal{E}$. The entire computation $\mathcal{E}$ is itself considered a region that encloses all other regions. For any node v, we say that a region A ***contains*** v if v is a node in A's dag. We say that v ***belongs to*** region A if A is the innermost region that contains v. Let $\texttt{regions}(\mathcal{E})$ denote the set of all regions for $\mathcal{E}$.

We assume that regions in $\mathcal{E}$ exhibit a canonical structure that satisfies the following assumptions. First, the (sub)dag for each region A contains a unique initial node $\texttt{root}(A)$ and a unique final node $\texttt{final}(A)$ such that all other nodes in the region are successors of $\texttt{root}(A)$ and predecessors of $\texttt{final}(A)$. Second, the regions are properly nested: if A contains one node from B, then it contains all nodes from B.

For a given region A, the (total) ***work*** $T_1(A)$ of A is the number of nodes contained in A. The ***span*** $T_\infty(A)$ of A is the number of nodes along a longest path from $\texttt{root}(A)$ to $\texttt{final}(A)$. The span represents the time it takes to execute a region on an infinite number of processors, assuming all nested regions are eliminated and flattened into the outer region (and ignoring any mutual-exclusion requirements for locked regions). For the full computation $\mathcal{E}$, we leave off the superscript and say that $T_1 = T_1(\mathcal{E})$ and span is $T_\infty = T_\infty(\mathcal{E})$.

We also define the work and span for a region A considering only nodes belonging to A. The ***region work*** $\tau_1(A)$ of A is the number of nodes in the dag belonging to A. For any path h through the graph $\mathcal{E}$, the ***path length for region*** A of h is the number of nodes u along path h which belong to A. The ***region span*** $\tau_\infty(A)$ of A is the maximum path length for A over all paths h from $\texttt{root}(A)$ to $\texttt{final}(A)$. Intuitively, the region span of A is the time to execute A on an infinite number of processors, assuming A's nested regions complete instantaneously. As an example, in the computation dag in Figure 4, region D has $T_1(D) = 13$, $\tau_1(D) = 7$, $T_\infty(D) = 8$, and $\tau_\infty(D) = 5$. For a computation $\mathcal{E}$ with parallel regions, the ***aggregate region span*** is

$$\widetilde{T}_\infty = \sum_{A \in \texttt{regions}(\mathcal{E})} \tau_\infty(A) \ .$$

Finally, in order to consider the contention due to short acquires of helper locks, we define the ***bondage*** b of a computation $\mathcal{E}$ as the total number of nodes corresponding to critical sections for short acquires of all helper locks.

The completion-time bound also depends on the number of regions and how regions are nested and connected to each other. The ***region graph*** for a computation $\mathcal{E}$ has a node for each region in $\mathcal{E}$ and an edge from region $A \in \texttt{regions}(\mathcal{E})$ to region $B \in \texttt{regions}(\mathcal{E})$ if some worker in region A calls either `start_region` for B or `help_region` for some lock connected to B. We say B is a ***child region*** of A if (A, B) is an edge in the region graph. We shall generally use the notation V for the number of regions in a region graph and E for the number of edges.

To state the space bounds, we need the following definition. For a region A, we define the ***serial space*** $\mathcal{S}_1(A)$ required by A, as the maximum stack space consumed by region A during the computation $\mathcal{E}$.

Statement of Bounds

Let T be the running time using HELPER of a computation $\mathcal{E}$ running on P processors. We prove that a computation $\mathcal{E}$ with V regions, E edges between regions in the region graph, work T_1, aggregate span $\widetilde{T}_\infty$, and bondage b runs in expected time

$$\mathrm{E}[T] = O(T_1/P + \widetilde{T}_\infty + E\ln(1+PV/E) + b). \tag{1}$$

Moreover, for any $\varepsilon > 0$, we prove that with probability at least $1-\varepsilon$, the execution time is

$$T = O(T_1/P + \widetilde{T}_\infty + E\ln(1+PV/E) + \lg(1/\varepsilon) + b). \tag{2}$$

Our space bounds are a generalization of the bounds in [4]. Let $\widetilde{\mathcal{S}}_1 = \sum_{A \in \texttt{regions}(\mathcal{E})} \mathcal{S}_1(A)$ where $\mathcal{S}_1(A)$ is the serial stack space required by region A. We prove that HELPER executes a computation $\mathcal{E}$ on P processors using at most $O(P\widetilde{\mathcal{S}}_1)$ space.

Interpretation and Discussion of Bounds

To understand what the completion-time bound in Equation (1) means, we can compare it to the completion-time bound for a computation without regions. The ordinary bound for randomized work stealing [4] says that the expected completion time is $O(T_1/P + T_\infty)$. The bound in Equation (1) exhibits three differences.

First, there is an additive term of $E\ln(1+PV/E)$. We shall show that $V - 1 \le E \le PV$. Therefore, in the best case, if $E = O(V)$, the term reduces to $V\ln P$. This case occurs when a computation has no contention on helper locks, i.e., when no `help_region` calls are made. In the worst case, if $E = PV$ this term is equal to PV. This worst case assumes that each worker assigned to a region enters from a different region, whereas we expect that most regions would have a limited number of entry points.

Even in the worst case, when the additive term is PV, if the number of parallel regions is small, then this term is insignificant compared to the other terms in the bound. Since parallel regions are meant to represent large critical sections, we expect V to be small in most programs. For example, in the hash-table example from Section 2, if we perform n insertions, only $O(\lg n)$ resizes occur during the execution. Furthermore, even if there are a large number of parallel regions, if each parallel region A is sufficiently large ($\tau_1(A) = \Omega(P^2)$) or long ($\tau_\infty(A) \geq \Omega(P)$), then we have $PV = O(T_1/P + \widetilde{T}_\infty)$, and the PV term is asymptotically absorbed by the other terms. These conditions seem reasonable, since we expect programmers to use parallel regions only for large critical sections. Programmers should generally use short acquires to protect small critical sections.

Second, HELPER completion time involves the term $\widetilde{T}_\infty$ instead of T_∞. If the number of parallel regions is small (as we expect) then the term $\widetilde{T}_\infty$ is generally close to T_∞. Even for programs with a large number of parallel regions, the $\widetilde{T}_\infty$ term does not slow down the execution if the parallel regions are sufficiently parallel. To understand why, look at both the work-stealing bound and the parallel-regions bound in terms of parallelism. The ordinary work-stealing bound means that the program gets linear speedup if $P = O(T_1/T_\infty)$. That is, the program gets linear speedup if the parallelism of the program is at least $\Omega(P)$. We can restate the HELPER bound as follows:

$$O\left(\frac{T_1}{P} + \widetilde{T}_\infty + PV\right) = O\left(\sum_{A \in \texttt{regions}(\mathcal{E})} \left(\frac{\tau_1(A)}{P} + \tau_\infty(A) + P\right)\right).$$

We can now see (ignoring the PV term) that HELPER provides linear speedup if the parallelism of each region is at least $\Omega(P)$. In the hash-table example, a region A that resizes a table of size k completely in parallel has span $\tau_\infty(A) = O(\lg k)$, and thus on most machines, the parallelism should greatly exceed the number of processors.

Third, we have the additive term b. In most programs, it is unlikely that the real completion time with HELPER will include all of the bondage. It is difficult to prove a tighter bound, however, since theoretically, there exist computations for which no runtime system could do any better. For example, if all lock acquires are for the same helper lock L, then, ignoring the $E\ln(1 + PV/E)$ term, our bound is asymptotically optimal. (No runtime system can execute the computation asymptotically faster.)

Comparison with Alternative Implementations

We now compare the bounds of our implementation of parallel regions with two other alternatives. The first option does not allow helping. When a worker p blocks on a lock L, it just waits until L to become available. Using this traditional implementation for locks, the completion time of a program with critical sections (either expressed as parallel regions or just expressed sequentially) can be $\Omega(T_1/P + \sum_{A \in \texttt{regions}(\mathcal{E})} \tau_1(A) + b)$. Notice that the second term is the sum of region work over all regions, as compared to Equation (1), which has the sum of region spans. Therefore, if the program has large (and highly parallel) critical sections (as in the hash-table example), then this implementation may run significantly slower than with helper locks.

Second, we can compare against an implementation where a worker that blocks on a lock suspends its current work and randomly work-steals. As the hash-table example from Section 2 illustrates, this implementation may use $\Omega(T_1)$ space. In contrast, HELPER uses $O(PT_\infty)$ space for this example, which is much smaller than T_1 for reasonable parallel programs.

5. ABSTRACT EXECUTION MODEL

This section formalizes the runtime system described in Section 3. We shall use this model in Section 6 to prove completion time bounds for HELPER. This model extends the model of Arora *et al.* in [2] by incorporating parallel regions and transitions of workers between regions.

Definitions

We first adopt some additional terminology. Let $\mathcal{E}$ be a computation. As in [2], we assume that each node $u \in \mathcal{E}$ has degree at most 2. Computation dags contain three types of nodes. A node u is a ***spawn node*** if u has in-degree 1 and out-degree 2, a ***sync node*** if u has in-degree 2 and out-degree 1, and a ***serial node*** if u has in-degree and out-degree 1. Without loss of generality, we assume that if a parallel region B is directly nested inside a parent region A, then $\texttt{root}(B)$ is immediately preceded by a serial node in A and $\texttt{final}(B)$ is immediately succeeded by a serial node in A.

For a region A, we say that a serial node $u \in A$ is a ***helper node*** if it marks a place after which HELPER may suspend execution of region A (because of a `start_region` or `help_region` call) to work on a node u' in a different region. For any helper node $u \in A$, we define the ***region successor*** of u as the unique node $v \in A$ where execution resumes when HELPER returns to A. When a helper node u corresponds to a `start_region` call for a region B directly nested inside A, then u is the immediate predecessor of $\texttt{root}(B)$ and v is the immediate successor of $\texttt{final}(B)$. When u corresponds to a `help_region` call to region B from region A, then v is simply the immediate successor node of the serial node u, which without loss of generality, we assume belongs to A. In both cases, we say that u is the ***region predecessor*** of v.

Execution

HELPER uses an execution model similar to the ones described in [2] and [4]. A node v is ***ready*** if all of v's predecessors have already been executed. Each worker maintains a ***deque*** of ready nodes. In any time step for which a worker has work available, the worker owns an ***assigned*** node, which it executes. In computations without helper locks or parallel regions, each worker p takes one of the following actions on each time step:

1. If p's assigned node is a spawn node u, then executing u makes two nodes ready. One of u's immediate successors is pushed onto p's deque, and the other immediate successor becomes p's assigned node.
2. If p's assigned node is a serial or sync node u, and executing u makes its immediate successor node v ready, then v becomes p's assigned node.
3. If p's assigned node is a serial or sync node u, but executing u does not make u's successor node v ready (because v is a sync node), then p removes the bottom node w from its deque and assigns it, if such a w exists. Otherwise, after the time step, p has an empty deque and no assigned node.
4. If p has no assigned node and an empty deque, then p becomes a ***thief***, chooses a ***victim*** deque uniformly at random, and attempts to steal work from the victim. If the victim's deque is not empty, p steals the deque's top node x, x becomes p's assigned node, and the steal attempt is considered ***successful***. Otherwise, the deque is empty and the steal attempt fails.

To support parallel regions in HELPER, we extend this execution model to incorporate multiple deques on each worker, as well as the transitions of workers between regions. In the extended model, each worker p owns an active deque $\texttt{activeDQ}(p)$. This active deque always contains p's currently assigned node. Each p may also own several inactive deques. An inactive deque is either empty, or it has a ***blocked node*** at the bottom which represents the node where execution will resume once that deque becomes active again.

Consider a worker p with an active deque $q_A = \texttt{activeDQ}(p)$ and active region $A = \texttt{region}(q)$. The extended model includes the following additional actions for p:

5. Suppose that p executes a helper node u belonging to region A, where u corresponds to a `start_region` call. In this case,

u's immediate successor in $\mathcal{E}$ is $\texttt{root}(B)$ for another region B. First, worker p creates a new active empty deque q_B with $\texttt{region}(q_B) = B$ and assigned node $\texttt{root}(B)$. Then, p changes $\texttt{activeDQ}(p)$ from q_A to q_B, and sets u's region successor as the blocked node for q_A.

6. Suppose that p executes a helper node $u \in A$, where u corresponds to a `help_region` call to B. First, p creates a new active empty deque q_B with $\texttt{region}(q_B) = B$ and no assigned node. Then, as in the previous case, p changes $\texttt{activeDQ}(p)$ from q_A to q_B and sets u's region successor as q_A's blocked node.
7. Suppose that p with active region A has no assigned node. Then, p attempts to steal from a randomly chosen victim deque $q \in \texttt{dqpool}(A)$. If q is not empty and the top node x of the deque is not a blocked node, then the steal attempt succeeds as before, i.e., p steals x, and x becomes p's assigned node. If q is empty and active (i.e., q has no child deque), then the steal fails.

   We call any deque $q \in \texttt{dqpool}(A)$ an ***exposed deque*** for A if either q has a blocked node on top or q is empty and inactive. An exposed deque always has a child deque. If p tries to steal from an exposed deque q with a child q' (where $B = \texttt{region}(q')$), then p enters B. It does so by creating a new active deque q_B in B's deque pool and changing its active deque to q_B. An exposed deque may cause workers assigned to A to enter B due to random work stealing.
8. Finally, if p executes $\texttt{final}(A)$, then each worker p_i with a deque $q_i = \texttt{dq}(p_i, A)$ must have $q_i = \texttt{activeDQ}(p_i)$. Each worker p_i changes $\texttt{activeDQ}(p_i)$ to $q_i' = \texttt{parent}(q_i)$. If q_i' has a blocked node v, then v becomes p_i's assigned node.

We call any step on which a worker p tries to enter a region (e.g., Cases 5 and 6 and sometimes Case 7) an ***entering step*** for p. Similarly, any step on which p executes Case 8 is called a ***leaving step*** for p. Otherwise, the step is a ***running step*** for p. For Case 7, on any step when p tries to steal from another worker p', and p' takes an entering step into A, we assume that p queues up and tries to enter A.

The extended execution model preserves the set of invariants enumerated in the following lemma. The lemma, which is stated without proof, can be proved by induction on the actions of the execution model.

LEMMA 1. *During a computation $\mathcal{E}$, consider a deque q owned by worker p. Let $v_1, v_2, \ldots, v_k$ be the nodes on q, arranged from bottom to top. Let u_i be an immediate predecessor of v_i if v_i is an unblocked node, or the region predecessor of v_i if v_i is a blocked node. The execution model maintains the following invariants:*

1. *For all i, node v_i belongs to region $A = \texttt{region}(q)$.*
2. *For all i, node u_i is unique.*
3. *For all unblocked nodes v_i, node u_i is a `spawn` node.*
4. *If v_i is blocked, then $i = 1$, that is, a blocked node must be at the bottom of a deque.*
5. *Deque q contains a blocked node v_i if and only if q is inactive, that is, $q \neq \texttt{activeDQ}(p)$.*
6. *For $i = 2, 3, \ldots, k$, node u_i is a predecessor of node u_{i-1}.*
7. *If q is active with assigned node w, then w is a successor of u_1.*

□

6. PROOF OF PERFORMANCE BOUNDS

This section proves the bounds on completion time and space explained in Section 4. We first prove the completion-time bound without considering the contention on short acquires. In other words, we assume that no worker ever blocks on a short acquire of a lock and prove a bound that omits the bondage term b from Equation (1). We remove the assumption at the end of this section.

Proof Outline for Completion Time

To bound the completion time, we account for each possible action that each processor (worker) p can take on each time step. Every processor step falls into one of the following categories:

- **Working**: p executes its assigned node.
- **Entering**: p waits as it tries enter a region A.
- **Leaving**: p waits as it tries to leave a region A or fails to steal from a worker p' which is leaving A.
- **Stealing**: p attempts to steal work randomly in a region A.

Let $\mathcal{E}$ be a computation executed on P workers, let V be the number of regions in $\mathcal{E}$, and let T_1 and $\widetilde{T}_\infty$ be the work and aggregate span of $\mathcal{E}$, respectively. Then, $\mathcal{E}$ has exactly T_1 working steps. For the deque-pool implementation described in Section 3, the entering cost is $O(\lg P)$ per worker, and once a region A completes, each worker in A's deque pool leaves within $O(\lg P)$ time. Therefore, the total number of entering and leaving steps is $O(PV \lg P)$.

We bound the number of steal attempts by partitioning them into three types which we bound individually. A ***contributing*** steal for region A is any steal attempt in A that occurs on a step when A has no exposed deques. A ***leaving*** steal for region A is any steal attempt which occurs while A has an exposed deque, and there exists a region B and a worker p such that B is a child of A, $\texttt{region}(\texttt{activeDQ}(p)) = B$, and p is executing a leaving step. A steal attempt in a region A which is neither a leaving nor contributing steal is considered an ***entering*** steal.

We follow Arora, Blumofe, and Plaxton [2] and use a potential-function argument to show that every region A has $O(P\tau_\infty(A))$ contributing steals in expectation, which implies that the expected total number of contributing steals is bounded by $O(P\widetilde{T}_\infty)$. One can bound the number of leaving steals by P times the number of time steps when any leaving step occurs. Since any worker p in A's deque pool leaves within $O(\lg P)$ time of A's completion, and no worker enters the same region twice, the total number of time steps when any worker can be leaving a region is at most $O(V \lg P)$. Therefore, the total number of leaving steals is $O(PV \lg P)$. Finally, we show that the total number of entering steals is at most $O(PE \ln(1 + PV/E))$.

Potential Function

Before defining the potential, we require two auxiliary definitions. For every node $u \in A$, the ***depth*** $d(u)$ of u is the maximum path length for region A over all paths from $\texttt{root}(A)$ to u. The ***weight*** of a region A is $w(A) = \tau_\infty(A) - d(A)$.[1]

As in [2], the weights of nodes along any deque strictly decrease from top to bottom.

LEMMA 2. *For any deque q owned by a worker p, let $v_1, v_2, \ldots, v_k$ be the nodes in q ordered from the bottom of the deque to the top, and let v_0 be the assigned node if $q = \texttt{activeDQ}(p)$. Then, we have $w(v_0) \leq w(v_1) < \cdots < w(v_k)$.*

PROOF. If the deque q is not empty, then either v_1 is a blocked node, or an assigned node v_0 exists. Invariants 4 and 5 from Lemma 1 imply that these two conditions are mutually exclusive. Define the u_i's as in Lemma 1.

On the one hand, suppose that v_1 is a blocked node and v_0 does not exist. Since we assume $\mathcal{E}$ is a series-parallel dag, the depth of any `spawn` node u is always 1 less than the depth of its two children. By Invariant 3, for all the unblocked nodes $v_2, v_3, \ldots, v_k$ in the deque, we have $d(u_i) = d(v_i) - 1$. Similarly, for a blocked node v_1, the region predecessor u_1 satisfies $d(u_1) = d(v_1) - 1$. Invariant 6 implies that u_i is a predecessor of u_{i-1} in $\mathcal{E}$, and hence

[1] In [2], depth and weight are defined in terms of an execution-dependent enabling tree. We can use this simpler definition, because we are only considering series-parallel computations.

$d(v_{i-1}) > d(v_i)$ for all $i = 2,3,\ldots,k$. Converting from depth to weight yields $w(v_1) < w(v_2) < \cdots < w(v_k)$.

On the other hand, suppose that v_0 does exist (and thus, q contains only unblocked nodes). Applying the same logic to the nodes on the deque as in the first case, we have $d(v_{i-1}) > d(v_i)$ for $i = 2,3,\ldots,k$. Invariant 7 implies that v_0 is a successor of u_1, and thus we have $d(v_0) > d(u_1) = d(v_1) - 1$, which implies $d(v_0) \geq d(v_1)$. Converting from depth to weight reverses the inequalities, yielding $w(v_0) \leq w(v_1) < \cdots < w(v_k)$. □

We now define the potential for nodes and extend it to deques and regions.

DEFINITION 1. *The **potential** of a node $u \in \mathcal{E}$ is*

$$\Phi(u) = \begin{cases} 3^{2w(u)-1} & \text{if } u \text{ is assigned,} \\ 3^{2w(u)} & \text{if } u \text{ is ready or blocked.} \end{cases}$$

We extend the potential to deques as follows. Let q be a deque belonging to worker p, and if q is active, let u be p's assigned node. Define the potential of q as

$$\Phi(q) = \begin{cases} \sum_{v \in q} \Phi(v) & \text{if } q \text{ is inactive,} \\ \Phi(u) + \sum_{v \in q} \Phi(v) & \text{if } q \text{ is active.} \end{cases}$$

We extend the potential to a region A as follows:

$$\Phi(A) = \sum_{q \in \texttt{dqpool}(A)} \Phi(q) \; .$$

LEMMA 3. *During a computation $\mathcal{E}$, the potential $\Phi(A)$ of any region $A \in \texttt{regions}(\mathcal{E})$ increases from 0 to $3^{2\tau_\infty(A)-1}$ when $\texttt{root}(A)$ becomes ready. At no other time does $\Phi(A)$ increase.*

PROOF SKETCH. To prove the lemma, one can check all the cases of the execution model. In general, actions that execute or assign nodes within A affect only $\Phi(A)$, and by the proof given in [2], these actions only decrease the potential. The most interesting case that HELPER introduces is when a worker p with active deque $q = \texttt{activeDQ}(p)$ executes an (assigned) helper node u from region $A = \texttt{region}(q)$ (Cases 5 and 6). In these cases, the region successor v of u is added to q as a blocked node. By definition of the region successor, we have $d(v) = d(u) + 1$, and hence, the potential decrease in region A is $3^{2w(u)-1} - 3^{2w(v)} = 2 \cdot 3^{2w(v)}$. In Case 5, worker p assigns $\texttt{root}(B)$ for the region B that was just started, increasing $\Phi(B)$ to $3^{2\tau_\infty(B)-1}$. In Case 6, $\Phi(B)$ is unchanged because p creates an empty deque for B. □

Contributing Steals

To bound the number of contributing steals in a region A, we divide steal attempts in A into rounds. The first round $R_1(A)$ in region A begins when $\texttt{root}(A)$ is assigned to some worker. A ***round*** $R_k(A)$ ends after at least P contributing steals in A have occurred in the round, or when A ends. Any round can have at most $2P-1$ contributing steals (if P steals occur in the last time step of the round). Therefore, a round has $O(P)$ contributing steal attempts in region A. We say $R_{k+1}(A)$ begins on the same time step of the next contributing steal in A after $R_k(A)$ has ended. A round in A may have many entering steals and there may be gaps of time between rounds.

First, we bound the number of rounds for a region A. At the beginning of $R_k(A)$, let $D_k(A)$ be the sum of potentials of all nonempty deques that are not exposed in A's deque pool, and let $E_k(A)$ be the sum of potentials due to empty or exposed deques. The potential of A at the beginning of round k can then be expressed as $\Phi_k(A) = D_k(A) + E_k(A)$.

LEMMA 4. *For any round $R_k(A)$ of a region $A \in \texttt{regions}(\mathcal{E})$, we have*

$$\Pr\{\Phi_k(A) - \Phi_{k+1}(A) \geq \Phi_k(A)/4\} \geq 1/4.$$

PROOF. We shall show that each of $D_k(A)$ and $E_k(A)$ decrease by at least a factor of $1/4$ with probability at least $1/4$. The lemma holds trivially for the last round of A, since the region completes.

Any deque that contributes potential to $D_k(A)$ has, at the beginning of the round, at least one node on top that can be stolen. By definition, every round, except possibly the last round, has at least P steal attempts. Thus, Lemma 8 from [2] allows us to conclude directly that $D_k(A)$ decreases by a factor of $1/4$ with probability $1/4$. Any entering steals that occur only reduce the potential further.

To show that $E_k(A)$ reduces by at least $1/4$, we use the definition of rounds and contributing steals, which imply that on the first time step of round k, region A has no exposed deques. Thus, any deque q that contributes to $E_k(A)$ must be empty. An empty deque q without an assigned node contributes nothing to $E_k(A)$. An empty deque q with an assigned node u reduces q's contribution to $E_k(A)$ by more than $1/4$, since u is executed in the round's first time step. □

LEMMA 5. *For any region $A \in \texttt{regions}(\mathcal{E})$, the expected number of rounds is $O(\tau_\infty(A))$, and the number of rounds is $O(\tau_\infty(A) + \ln(1/\varepsilon))$ with probability at least $1-\varepsilon$.*

PROOF. The proof is analogous to Theorem 9 in [2]. Call round k ***successful*** if $\Phi_k(A) - \Phi_{k+1}(A) \geq \Phi_k(A)/4$, i.e., the potential decreases by at least a $1/4$ fraction. Lemma 4 implies that $\Pr\{\Phi_{k+1}(A) \leq 3\Phi_k(A)/4\} \geq 1/4$, i.e., a round is successful with probability at least $1/4$. The potential for region A starts at $3^{2\tau_\infty(A)-1}$, ends at 0, and is always an integer. Thus, a region A can have at most $8\tau_\infty(A)$ successful rounds. Consequently, the expected number of rounds needed to finish A is at most $32\tau_\infty(A)$. For the high probability bound, one can use Chernoff bounds as in [2]. □

Entering Steals

To bound the number of entering steals, we require some definitions. Intuitively, we divide the entering steals for a particular region A into "intervals", and subdivide intervals into "phases". For every region A, we divide the time steps when A is active into ***entering intervals***, which are separated by leaving steps for child regions B. More precisely, entering interval 1 for A, denoted $I_1(A)$, begins with the first entering steal in A and ends with the next leaving step belonging to any region B such that (A,B) is an edge in the region graph for $\mathcal{E}$, or if A completes. Similarly, $I_k(A)$ begins with the first entering steal in A after $I_{k-1}(A)$ completes.

We also subdivide an interval $I_k(A)$ into ***entering phases***, separated by successful entering steals in A. In general, phase j of $I_k(A)$, denoted $I_{k,j}(A)$, begins with the first entering steal of $I_k(A)$ after phase $j-1$ and ends with the next successful entering steal, or at the end of $I_k(A)$. [2] We say that a phase $I_{k,j}(A)$ has ***rank*** j. Define an entering phase as ***complete*** if it ends with a successful entering steal. Every interval has at most one incomplete phase (the last).

Intuitively, entering intervals and phases are constructed so that during an interval for A, the number of exposed deques only increases, and the probability of a successful entering steal increases with the rank of the current phase.

LEMMA 6. *For any entering interval $I_k(A)$ for a region $A \in \texttt{regions}(\mathcal{E})$,*

[2] Note that the end of $I_k(A)$ and the beginning of $I_{k+1}(A)$ occur on different time steps, but the end of phase j and beginning of phase $j+1$ within an interval can occur within the same time step. If multiple workers try to steal from the same exposed deque in a time step, they all succeed, and multiple entering phases end in that time step.

1. *interval $I_k(A)$ has at most $P-1$ phases; and*
2. *during any phase of rank j, the probability that a given steal attempt succeeds is at least j/P.*

PROOF. At the beginning of the interval, there is at least one exposed deque in A, and therefore, at least one worker has moved from A to some child region of A. Every complete phase ends with a successful entering steal and a successful entering steal causes a worker to move from A to some child region. Therefore, after j phases, at least $j+1$ workers have moved to some child region. Moreover, an interval $I_k(A)$ ends with any leaving step for any child region of A. Therefore, no worker re-enters region A from its child region during an interval. Due to these two facts, after $P-1$ complete phases, no processors are working in region A and there can be no more entering steals. On the other hand, if the last phase is incomplete, then the number of complete phases is less than $P-1$. In either case, the interval has at most $P-1$ phases.

No exposed deque is eliminated during an interval since no processor reenters A and thus, the number of exposed deques in A never decreases. At the beginning of $I_{k,1}(A)$, the first phase of every interval, A has at least one exposed deque. Therefore, during phase 1, the probability of any entering steal succeeding is at least $1/P$. As we argued above, after $j-1$ phases complete, at least j workers have entered a child region and each of these workers leave behind an exposed deque in region A (since their deque $\mathtt{dq}(p,A)$ is empty). Hence in phase j, interval $I_{k,j}(A)$ has at least j exposed deques[3] and the probability of hitting one of these exposed deques on any steal attempt is at least j/P. □

LEMMA 7. *Let $\mathcal{E}$ be a computation executed on P processors whose region graph has V regions and E edges, and let r_m be the number of phases of rank m. The following hold:*

1. $V-1 \le E \le PV$.
2. *The number of entering intervals over all regions is at most E.*
3. $E \ge r_1 \ge r_2 \cdots \ge r_{P-1}$.
4. *The total number of complete entering phases over all regions is at most $V(P-1)$. The total number of incomplete entering phases is at most E.*
5. *For any integer $K \ge r_1$, let $\alpha(K) = \lceil \sum_{j=1}^{P-1} r_j/K \rceil$. Then, for any nonincreasing function f, we have*

$$\sum_{j=1}^{P-1} r_j f(j) \le K \sum_{j=1}^{\alpha(K)} f(j).$$

PROOF. (1) Since only P workers total can enter a region B, and in the worst case each worker enters along a different edge, every region B can have in-degree at most P, and thus we have $E \le PV$. Moreover, every region except $\mathcal{E}$ also has in-degree at least 1, which implies that $V-1 \le E$.

(2) For any region A, let d_A be the out-degree of A in the region graph for $\mathcal{E}$. Intervals for region A end only when some child B completes and takes a leaving step. Also, the time after the leaving step for the last child region B does not form an interval because there can be no more entering steals in A. Thus, A has at most d_A intervals. Summing over all regions, we can have at most E intervals total.

(3) By construction, every interval can have at most one phase of a given rank m, and thus we have $E \ge r_1$. Also, an interval can have a phase of rank $m+1$ only if it has a phase of rank m, which implies $r_m \ge r_{m+1}$.

(4) Every complete entering phase has a successful entering steal. We can have at most $P-1$ successful entering steals for each region A, since one worker enters the region when A is started and at most $P-1$ can enter through stealing. Every interval can have at most 1 incomplete phase.

(5) The quantity $\sum_{j=1}^{P-1} r_j f(j)$ can be viewed as the sum of entries in a E by $P-1$ grid, where row z contains phases for the zth interval, the jth column corresponds to phases of rank j, and entry (z,j) has value $f(j)$ if interval z has a phase of rank j, or 0 otherwise. This grid contains at most r_j nonzero entries in each column, and $\sum_j r_j$ entries total. Since the function $f(j)$ and r_j are both nonincreasing and $K \ge r_1$, conceptually, by moving entries left into smaller columns, we can compress the nonzero entries of the grid into a compact K by $\alpha(K) = \lceil \sum_j r_j/K \rceil$ grid without decreasing the sum. The value of the compact grid is an upper bound for the value of the original grid. The compact grid has at most $\alpha(K)$ columns, with each column j having value at most $Kf(j)$, giving the desired bound. □

We can now bound the expected number of entering steals.

THEOREM 8. *For a computation $\mathcal{E}$ executed on P processors whose region graph has V regions and E edges, the expected number of entering steal attempts is $O(PE\ln(1+PV/E))$.*

PROOF. Suppose that $\mathcal{E}$ requires r_m phases of rank m. Number the intervals of $\mathcal{E}$ arbitrarily, and let $Q(z,j)$ be the random variable for the number of entering steals in phase j of the zth interval, or 0 if the interval or phase does not exist. By Lemma 7, there can be at most E intervals. Thus, the total number Q of entering steals is given by

$$Q = \sum_{z=1}^{E} \sum_{j=1}^{P-1} Q(z,j).$$

Lemma 6 implies that $E[Q(z,j)] \le P/j$, since each entering steal in the phase succeeds with probability at least j/P. Thus, linearity of expectation gives us

$$E[Q] = \sum_{z=1}^{E} \sum_{j=1}^{P-1} E[Q(z,j)] \le \sum_{z=1}^{E} \sum_{j=1}^{P-1} \frac{P}{j} = \sum_{j=1}^{P-1} r_j \left(\frac{P}{j}\right). \tag{3}$$

We can apply Fact 5 of Lemma 7 with $K=E$ and $f(j)=1/j$, since $E \ge r_1$. Then, we have $\alpha(K) = \lceil \sum_j r_j/E \rceil \le 1 + \lceil PV/E \rceil$, since $\sum_j r_j$ is the total number of phases, which is bounded by $(P-1)V+E$. Thus, we have

$$E[Q] \le PE \sum_{j=1}^{\alpha(K)} \frac{1}{j} = PEH_{\alpha(K)} ,$$

where $H_n = \sum_{i=1}^{n} 1/i = O(\ln n)$ is the nth harmonic number, which completes the proof. □

We can also bound the number of entering steals with high probability.

THEOREM 9. *For a computation $\mathcal{E}$ executed on P processors whose region graph has V regions and E edges, the number of entering steal attempts is $O\left(PE\ln(1+PV/E) + P\ln^2 P + P\ln(1/\varepsilon)\right)$ with probability at least $1-\varepsilon$.*

PROOF SKETCH. Let r_j be the number of phases of rank j in $\mathcal{E}$. One can bound the number of entering steals as a function of the r_j. Divide the entering steals into $\lceil \lg P \rceil$ classes of entering steals, where any entering steal in a phase of rank j is assigned to class $m = \lfloor \lg j \rfloor + 1$. Let $R_m = \sum_{j=2^{m-1}}^{2^m-1} r_j$ be the number of

[3] Phase j might have more than j such deques, since `help_region` calls into A or steals within A can exposes new deques.

phases in class m. By Lemma 6, each class-m entering steal succeeds with probability $\beta \geq 2^{m-1}/P$. Thus, one can use Chernoff bounds to show the probability of having more than $O(PR_m/2^m + P\ln(\lceil \lg(P) \rceil/\varepsilon)/2^m)$ class-m steals is less than $\varepsilon/\lceil \lg P \rceil$. Union-bound over all classes to obtain the final bound. □

Bounding Completion Time

We can now bound the completion time of a computation.

THEOREM 10. *Let $\mathcal{E}$ be a computation executed on P workers, let V be the number of regions in $\mathcal{E}$, and let E be the number of edges in $\mathcal{E}$'s region graph. If $\mathcal{E}$ has work T_1 and aggregate span $\widetilde{T}_\infty$, then HELPER completes $\mathcal{E}$ in expected time $O(T_1/P + \widetilde{T}_\infty + E\ln(1+PV/E))$. Moreover, for any $\varepsilon > 0$, the completion time is $O(T_1/P + \widetilde{T}_\infty + E\ln(1+PV/E) + \ln^2 P + \lg(1/\varepsilon))$ with probability at least $1-\varepsilon$.*

PROOF. On every time step, each of the P processors is working, entering, leaving, or stealing. Executing a computation requires exactly T_1 work steps. The number of entering steps and leaving steps is at most $O(PV\lg P)$, since every worker waits at most $O(V)$ times (at most once for entering and leaving every region A), and each worker spends only $O(\lg P)$ waiting each time.

To bound the number of steals, by Lemma 4, we expect $O(P\widetilde{T}_\infty)$ contributing steals, and by Theorem 8, we have $O(E\ln(1+PV/E))$ entering steals. Finally, the number of leaving steals is bounded by P times the number of time steps that any worker spends on a leaving step. Thus, we have only $O(PV\lg P)$ leaving steals.

Since P workers are active on every step, adding up the bounds on all the steps and dividing by P gives us the expectation bound. To obtain a high-probability bound, we choose $\varepsilon' = \varepsilon/2$ for both Lemma 4 and Theorem 9, and then union-bound over the two cases. □

Time Bound with Short Acquires

We now add the contention on short acquires into the analysis. It is difficult to prove interesting bounds for computations with regular locks. Since short acquires of helper locks are no different than regular lock acquires (a program with only short acquires behaves in exactly the same way as a program with regular (non-helper) locks), this difficulty in proving bounds extends to short acquires. Here, we present a simple bound for completeness.

As defined in Section 4, the bondage b of a computation $\mathcal{E}$ represents the amount of computation within critical sections protected by short acquires. In HELPER, when a worker p holds a helper lock L and another worker p' blocks trying to acquire L, worker p' waits. In this case, we say that p' takes a ***waiting*** step.

LEMMA 11. *For a computation $\mathcal{E}$ with bondage b executed on P processors, the total number of waiting steps (due to short acquires) is at most Pb.*

PROOF. When any worker takes a waiting step, there exists a p holding a lock L and executing some node in a critical section protected by a short acquire. Therefore, the remaining bondage decreases by at least 1 during that time step. In the worst case, all $P-1$ other workers might be taking waiting steps during that time step, all waiting on lock L. □

Adding in waiting steps for each worker, the completion time bound with short acquires adds the bondage b to the bound in Theorem 10.

Space

The following theorem bounds stack space usage.

THEOREM 12. *For a computation $\mathcal{E}$, let $\mathcal{S}_1(A)$ be serial stack space required to execute a region $A \in regions(\mathcal{E})$, and let $\widetilde{\mathcal{S}}_1 = \sum_{A \in regions(\mathcal{E})} \mathcal{S}_1(A)$. Then, HELPER executes $\mathcal{E}$ on P processors using at most $O(P\widetilde{\mathcal{S}}_1)$ space.*

PROOF. At any point fixed in time, consider the tree T of active stack frames for the computation $\mathcal{E}$. For any region A, let T_A be the subset of T which consists of only frames which belong to A.

The only time a worker p stops working on its current active frame f in region A without completing f is if p enters a new region B. Also, a worker p can only enter a region A once. Using these two facts, one can show that the set T_A is, in actuality, a tree with at most k leaves, where k is the number of workers which have entered A. This fact is a generalization of the busy-leaves property from [4] applied only to nodes of the specific region A. Thus, A uses at most $O(k\mathcal{S}_1(A))$ stack space for T_A. In the worst case, all P workers enter every region $A \in \texttt{regions}(\mathcal{E})$, and we must sum the space over all regions, giving us a space usage of $\widetilde{\mathcal{S}}_1 = \sum_{A \in \texttt{regions}(\mathcal{E})} \mathcal{S}_1(A)$. □

7. PROTOTYPE IMPLEMENTATION

This section describes the HELPER prototype, which we created by modifying Cilk [6]. In this section, we discuss how we implement deque chains and deque pools, the two major additions that HELPER makes to the Cilk runtime.

Deque Chains

To implement parallel regions, we must conceptually maintain a chain of deques for each worker p. In ordinary Cilk, each deque is represented by pointers into a ***shadow stack***, a per-worker stack which stores frames corresponding to Cilk functions. HELPER maintains the entire deque chain for a worker on that worker's shadow stack.

Normally, Cilk uses the THE protocol described in [6] to manage deques. Each deque consists of three pointers that point to slots in the shadow stack. The ***tail pointer*** T points to the first empty slot in the stack. When a worker pushes and pops frames onto its own deque, it modifies T. The ***head pointer*** H points to the frame at the top of the deque. When other workers steal from this deque, they remove the frame pointed to by H and decrement H. Finally, the ***exception pointer*** E represents the point in the deque above which the worker should not pop. If a worker working on the tail end encounters $E > T$, some exceptional condition has occurred, and control returns to the Cilk runtime system.

In order to avoid the overhead of allocating and deallocating shadow stacks at runtime, HELPER maintains the entire chain of deques for a given worker p on the same shadow stack, as shown in Figure 5. Each deque for a given worker p maintains its own THE pointers, but all point into the same shadow stack. When a worker enters a region, it only needs to create the THE pointers for a new deque and set all these pointers equal to the to the tail pointer for the parent deque in the shadow stack. The correctness of this implementation relies on the property that two deques in the same shadow stack (for a worker p) can not grow and interfere with each other. This property holds for two reasons. First, in HELPER every worker p works locally only on its bottom active deque $\texttt{activeDQ}(p)$, and thus, only the tail pointer T of the active deque at the bottom of the chain can grow downward. Second, for any deque, the head H never grows upward, since H only changes when steals remove frames.

Figure 5 shows an example arrangement of a deque chain on a worker p's stack. In this example, no worker has stolen any frames from region B, whereas two frames have been stolen from region C. The deque $\texttt{dq}(p, E)$ is empty, and $\texttt{activeDQ}(p) = \texttt{dq}(p, F)$.

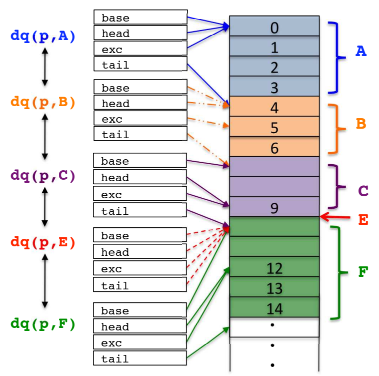

Figure 5. A chain of deques for a given worker p. Each deque $\mathtt{dq}(p,A)$ consists of pointers (tail, head, and exception) into p's stack of Cilk frames. For clarity, we show a pointer for the base of each deque q, although in practice, q's base always equals the tail pointer of q's parent deque.

Implementation of Deque Pools

The HELPER prototype implements a deque pool as a single array of deques (i.e., THE pointers). An array of P slots is statically allocated when the user creates a helper lock. When a region is active, one or more slots of this array are occupied by workers. Instead of dedicating a slot for every worker p as described in Section 3, however, our prototype maintains a packed array. If k workers are assigned to a region, the first k slots of the deque-pool array contains those workers. It also maintains a shared counter to track whether the pool is empty. Theoretically, with this packed array implementation, each worker p might spend $\Theta(P)$ time entering and leaving (as opposed to $O(\lg P)$ with the sparse array described in Section 3), if there is worst-case contention and p waits for all other workers. The contention is unlikely to be this bad in practice, however. Moreover, even with worst-case contention, using this scheme does not change the worst-case theoretical bounds. The bound remains the same as when E = PV, and the space bound is not affected. Also, with this scheme, workers may spend less time finding work in practice, because if work is available to be stolen, each steal succeeds with probability at least $1/k$, instead of $1/P$.

8. EXPERIMENTAL RESULTS

This section presents a small experimental study of an implementation of a resizable hash table using the HELPER prototype. Although the hash-table implementation is not optimized, these results suggest that HELPER is not merely a theoretical construct and that a practical and efficient implementation of HELPER may well be feasible.

Hash-Table Implementation

The resizable hash table maintains an array of pointers for hash buckets, where each bucket is implemented as a linked list. The table supports search and an atomic `insert_if_absent` function. A search or insertion locks the appropriate bucket in the table. When an insertion causes the chain in a bucket to overflow beyond a certain threshold, it atomically increments a global counter for the table. When more than a constant fraction of the buckets have overflowed, the insertion triggers a resize operation.

The resize operation sets a flag to indicate that a resize operation is in progress. It then acquires all the bucket locks, scans the buckets to compute the current size of the table, doubles the size of the table until the density of the table is below a certain threshold, allocates

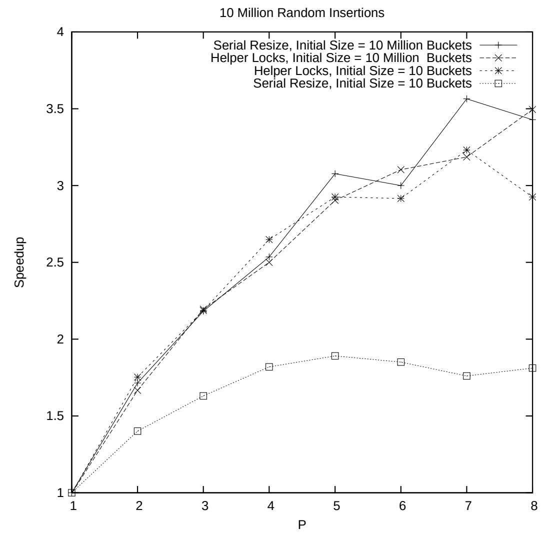

Figure 6. Results from an experiment inserting $n = 10^7$ random keys into a concurrent resizable hash table. Speedup is normalized to the runtime of the hash table with the same initial size and serial resize for $P = 1$. For tables with initial size of 10 and n buckets, the runtime was 8.66 s and 3.60 s, respectively. Each data point represents the average of 5 runs with the same parameters. This experiment was run on a two-socket quad-core (3.16-GHz Intel Xeon X5460) machine with 8 GB RAM.

a new array for buckets, and rehashes the elements from the old buckets into new buckets. We parallelized the lock acquires, the size computation, and the insertions into the new table.

We implemented two flavors of the hash table. The first flavor performs the resize operation serially, and the second spawns the resize operation as a parallel region protected by a resize region helper lock. Each bucket lock functions as a short helper lock linked to this resize lock. If the resize lock is held, then an attempt by a worker to acquire a bucket lock causes the worker to help resize.

Our benchmark performs n `insert_if_absent` operations on the hash table by spawning 20 functions, with each function performing $n/20$ inserts serially.[4] Keys are chosen uniformly at random based on a deterministic seed chosen for each of the 20 functions. Since each key k is random, the benchmark uses the simple hash function of taking k modulo the number of buckets in the hash table.

Experimental Results

Figure 6 shows results from performing insertions into the resizable hash table. We ran two versions of the experiment: one that contains no resize operations and one that does. In both experiments, the number of insertions was $n = 10^7$.

In the first experiment, the table began with 10^7 buckets, and thus, with 10^7 insertions, no resize operations were triggered. In this experiment, the implementations with both serial and parallel resize were comparable, and both provided a speedup of about 3.5 on 8 processors. These results indicate that the overhead of using a helper lock in this application is relatively small. In the second experiment, the table began with 10 buckets, and the table size repeatedly doubled on resizes. In this case, the implementation that uses a parallel resize operation with helper locks provided speedup

[4] The number 20 was chosen arbitrarily to be a number larger than P.

of about 3. In contrast, the implementation that used the serial resize provided speedup of at most 2. This experiment suggests helper locks have potential utility.

The plots in Figure 6 for the two experiments (10^7 versus 10 buckets) should not be directly compared to each other. The serial table that does not resize ran about 2.4 times faster than the serial hash table that does resize. This additional factor is approximately consistent with the amortized cost of table doubling. Theoretically, every insertion into a resizable table pays about 3 times the cost of a normal insertion: once for the original insertion, once to move the item when the table is expanded, and once to move another item that has already been moved once.

Our current hash-table implementation does not appear to scale beyond 4 processors, even when no resizes occur. Thus, additional work is needed to improve scalability in this benchmark. In practice, one hopes that a program exhibits a more realistic and scalable mix of concurrent operations (i.e., a combination of searches and insertions). Our primary goal for the benchmark, however, was to test and evaluate the feasibility of the HELPER prototype, and thus, we tried to trigger resize operations as often as possible.

9. CONCLUSION

We conclude by briefly reviewing some related work and discussing future research directions.

OpenMP uses a `parallel` construct to support nested parallelism [11]. HELPER exhibits some similarities to the implementation of this construct, although the design goals differ. For convenience in implementation, in the HELPER prototype, as in OpenMP, every parallel region has one (worker) thread which is the first to enter the region and which is guaranteed to resume execution after the region completes. Unlike in OpenMP, however, the number of workers is not fixed when a region begins. Additional workers can enter the region, either through random work stealing or because they are blocked on the lock for the region.

Cooperative techniques, where one thread helps another thread complete its work, have previously been proposed in a variety of contexts. In the context of nonblocking algorithms, researchers [3, 8, 13] describe algorithms where threads cooperate to complete an operation when they would otherwise block for synchronization. In the area of databases, Lim, Ahn, and Kim [10] describe a concurrent B^{link} tree algorithm which uses cooperative locking to handle nodes with concurrent underflow.

Both our implementation and experimental results can be improved to provide a more thorough evaluation of HELPER. For example, one avenue for future work is to implement a more sophisticated hash table using helper locks for resizing and to compare it to other optimized concurrent hash-table implementations. Most concurrent hash tables (e.g., hopscotch hashing by Herlihy, Shavit and Tzafrir [7]) are optimized for the case when most operations are queries. It would be interesting to explore whether one can efficiently parallelize resize operations using helper locks for times when updates are frequent without affecting the performance in the common case where queries dominate. Also, we would like to identify applications which might benefit from helper locks and benchmark these applications on an improved prototype of HELPER.

It appears to be difficult to improve the theoretical guarantees on completion time when programs can use helper locks and parallel regions arbitrarily. It would be interesting to see if real applications might use helper locks in a more restricted fashion, possibly allowing us to prove a stronger bound on completion time.

We plan to explore other applications of parallel regions. In our design, parallel regions have their own deque pool and are scheduled independently of other parts of the computation. Therefore, they could be used for purposes other than helper locks. For example, one might conceivably allow different regions to operate under different (specifically tailored) scheduling policies. Alternatively, one might design a more locality-aware runtime system, for example, by assigning workers that share a cache to the same region so that they are more likely to steal work from each other.

Acknowledgments

We would like to thank I-Ting Angelina Lee for helpful comments and discussions.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchango, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc., March 2008. URL `http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf`.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, 1998.

[3] G. Barnes. A method for implementing lock-free shared-data structures. In *SPAA '93: Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, New York, NY, USA, 1993. ACM. ISBN 0-89791-599-2.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[5] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: an experimental language for high productivity programming of scalable systems. In *Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*, Feb. 2005. Held in conjunction with the *Eleventh Symposium on High Performance Computer Architecture*.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[7] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *DISC '08: Proceedings of the 22nd International Symposium on Distributed Computing*, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC '94: Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, New York, NY, USA, 1994. ACM.

[9] C. E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, 2009. ACM.

[10] S.-C. Lim, J. Ahn, and M. H. Kim. A concurrent B^{link}-tree algorithm using a cooperative locking protocol. In *Lecture Notes in Computer Science*, volume 2712, pages 253–260. Springer Berlin / Heidelberg, 2003.

[11] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[12] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly, 2007.

[13] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *PODS '92: Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 212–222, New York, NY, USA, 1992. ACM.