On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs

Michael A. Bender Jeremy T. Fineman Seth Gilbert Charles E. Leiserson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

Abstract

A key capability of data-race detectors is to determine whether one thread executes logically in parallel with another or whether the threads must operate in series. This paper provides two algorithms, one serial and one parallel, to maintain series-parallel (***SP***) relationships "on the fly" for fork-join multithreaded programs. The serial ***SP-order*** algorithm runs in $O(1)$ amortized time per operation. In contrast, the previously best algorithm requires a time per operation that is proportional to Tarjan's functional inverse of Ackermann's function. SP-order employs an order-maintenance data structure that allows us to implement a more efficient "English-Hebrew" labeling scheme than was used in earlier race detectors, which immediately yields an improved determinacy-race detector. In particular, any fork-join program running in T_1 time on a single processor can be checked on the fly for determinacy races in $O(T_1)$ time. Corresponding improved bounds can also be obtained for more sophisticated data-race detectors, for example, those that use locks.

By combining SP-order with Feng and Leiserson's serial ***SP-bags*** algorithm, we obtain a parallel SP-maintenance algorithm, called ***SP-hybrid***. Suppose that a fork-join program has n threads, T_1 work, and a critical-path length of T_∞. When executed on P processors, we prove that SP-hybrid runs in $O((T_1/P + PT_\infty)\lg n)$ expected time. To understand this bound, consider that the original program obtains linear speed-up over a 1-processor execution when $P = O(T_1/T_\infty)$. In contrast, SP-hybrid obtains linear speed-up when $P = O(\sqrt{T_1/T_\infty})$, but the work is increased by a factor of $O(\lg n)$.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; E.1 [**Data Structures**]: distributed data structures; G.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*.

This research was supported in part by the Singapore-MIT Alliance, Sandia National Laboratories, and NSF grants ACI-032497, EIA-0112849, CCR-0208670, ITR-0121277, and AFOSR #F49620-00-1-0097.

Michael Bender is a Visiting Scientist at MIT CSAIL and Assistant Professor at the State University of New York at Stony Brook.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'04, June 27–30, 2004, Barcelona, Spain.
Copyright 2004 ACM 1-58113-840-7/04/0006 ...$5.00.

General Terms

Algorithms, Theory, Verification.

Keywords

Amortized analysis, algorithm, Cilk, data race, data structure, dynamic set, fork-join, graph, least common ancestor, locking, multithreading, mutual exclusion, on the fly, order maintenance, parallel computing, parse tree, race detection, series-parallel, SP-bags, SP-hybrid, SP-order, thread, trace, tree, work stealing.

1 Introduction

This paper shows how to maintain the series-parallel (***SP***) relationships between logical threads in a multithreaded program "on the fly." This problem arises as the principal data-structuring issue in dynamic data-race detectors [13, 19, 20, 26, 27]. In this paper, we show that for fork-join programming models, such as MIT's Cilk system [11, 21, 28], this data-structuring problem can be solved asymptotically optimally. We also give an efficient parallel solution to the problem.

The execution of a multithreaded program can be viewed as a directed acyclic graph, or ***computation dag***, where nodes are either ***forks*** or ***joins*** and edges are ***threads***. Such a dag is illustrated in Figure 1. A fork node has a single incoming edge and multiple outgoing edges. A join node has multiple incoming edges and a single outgoing edge. Threads (edges) represent blocks of serial execution.

For fork-join programming models, where every fork has a corresponding join that unites the forked threads, the computation dag has a structure that can be represented efficiently by a ***series-parallel (SP) parse tree*** [20]. In the parse tree each internal node is either an ***S-node*** or a ***P-node*** and each leaf is a thread of the dag.[1] Figure 2 shows the parse tree corresponding to the computation dag from Figure 1. If two subtrees are children of the same S-node, then the parse tree indicates that (the subcomputation represented by) the left subtree executes before (that of) the right subtree. If two subtrees are children of the same P-node, then the parse tree indicates that the two subtrees execute logically in parallel.

An SP parse tree can be viewed as an *a posteriori* execution of the corresponding computation dag, but "on-the-fly" data-race detectors must operate while the dag, and hence the parse tree, is unfolding dynamically. The way that the parse tree unfolds depends on a scheduler, which determines which threads execute where and when on a finite number of processors. A partial execution corresponds to a subtree of the parse tree that obeys the series-parallel relationships, namely, that a right subtree of an S-node cannot be

[1] We assume without loss of generality that all SP parse trees are full binary trees, that is, each internal node has exactly two children.

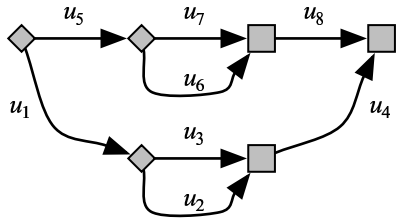

Figure 1: A dag representing a multithreaded computation. The edges represent threads, labeled $u_0, u_1, \ldots u_8$. The diamonds represent forks, and the squares indicate joins.

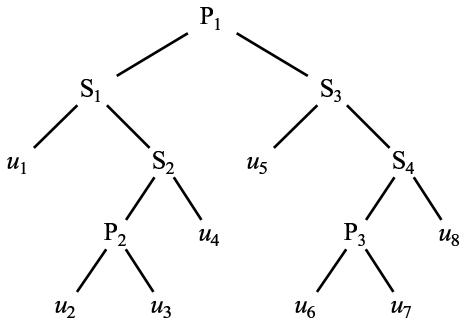

Figure 2: The parse tree for the computation dag shown in Figure 1. The leaves are the threads in the dag. The S-nodes indicate series relationships, and the P-nodes indicate parallel relationships.

present unless the corresponding left subtree has been fully elaborated. Both subtrees of a P-node, however, can be partially elaborated. In a language like Cilk, a serial execution unfolds the parse tree in the manner of a left-to-right walk. For example, in Figure 2, a serial execution executes the threads in the order of their indices.

A typical serial, on-the-fly data-race detector simulates the execution of the program as a left-to-right walk of the parse tree while maintaining various data structures for determining the existence of races. The core data structure maintains the series-parallel relationships between the currently executing thread and previously executed threads. Specifically, the race detector must determine whether the current thread is operating logically in series or in parallel with certain previously executed threads. We call a dynamic data structure that maintains the series-parallel relationship between threads an ***SP-maintenance*** data structure. The data structure supports insertion, deletion, and ***SP queries***: queries as to whether two nodes are logically in series or in parallel.

The Nondeterminator [13,20] race detectors use a variant of Tarjan's [30] least-common-ancestor algorithm, as the basis of their SP-maintenance data structure. To determine whether a thread u_i logically precedes a thread u_j, denoted $u_i \prec u_j$, their ***SP-bags algorithm*** can be viewed intuitively as inspecting their least common ancestor $\mathrm{lca}(u_i, u_j)$ in the parse tree to see whether it is an S-node with u_i in its left subtree. Similarly, to determine whether a thread u_i operates logically in parallel with a thread u_j, denoted $u_i \parallel u_j$, the SP-bags algorithm checks whether $\mathrm{lca}(u_i, u_j)$ is a P-node. Observe that an SP relationship exists between any two nodes in the parse tree, not just between threads (leaves).

For example, in Figure 2, we have $u_1 \prec u_4$, because $S_1 = \mathrm{lca}(u_1, u_4)$ is an S-node and u_1 appears in S_1's left subtree. We also have $u_1 \parallel u_6$, because $P_1 = \mathrm{lca}(u_1, u_6)$ is a P-node. The (serially executing) Nondeterminator race detectors perform SP-maintenance operations whenever the program being tested forks, joins, or accesses a shared-memory location. The amortized cost for each of these operations is $O(\alpha(v, v))$, where α is Tarjan's functional inverse of Ackermann's function and v is the number of shared-memory locations used by the program. As a consequence, the asymptotic running time of the Nondeterminator is $O(T_1\alpha(v, v))$, where T_1 is the running time of the original program on 1 processor.

Algorithm	Space per node	Time per	
		Thread creation	Query
English-Hebrew [27]	$\Theta(f)$	$\Theta(1)$	$\Theta(f)$
Offset-Span [26]	$\Theta(d)$	$\Theta(1)$	$\Theta(d)$
SP-Bags [20]	$\Theta(1)$	$\Theta(\alpha(v,v))$	$\Theta(\alpha(v,v))$
SP-Order	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

f = number of forks in the program
d = maximum depth of nested parallelism
v = number of shared locations being monitored

Figure 3: Comparison of serial, SP-maintenance algorithms. The running times of the English-Hebrew and offset-span algorithms are worst-case bounds, and the SP-bags and SP-order algorithms are amortized. The function α is Tarjan's functional inverse of Ackermann's function.

The SP-bags data structure has two shortcomings. The first is that it slows the asymptotic running time by a factor of $\alpha(v, v)$. This factor is nonconstant in theory but is nevertheless close enough to constant in practice that this deficiency is minor. The second, more important shortcoming is that the SP-bags algorithm relies heavily on the serial nature of its execution, and hence it appears difficult to parallelize.

Some early SP-maintenance algorithms use labeling schemes without centralized data structures. These labeling schemes are easy to parallelize but unfortunately are much less efficient than the SP-bags algorithm. Examples of such labeling schemes include the ***English-Hebrew*** scheme [27] and the ***offset-span*** scheme [26]. These algorithms generate labels for each thread on the fly, but once generated, the labels remain static. By comparing labels, these SP-maintenance algorithms can determine whether two threads operate logically in series or in parallel. One of the reasons for the inefficiency of these algorithms is that label lengths increase linearly with the number of forks (English-Hebrew) or with the depth of fork nesting (offset-span).

Results

In this paper we introduce a new SP-maintenance algorithm, called the ***SP-order*** algorithm, which is more efficient than the SP-bags algorithm. This algorithm is inspired by the English-Hebrew scheme, but rather than using static labels, the labels are maintained by an order-maintenance data structure [10, 15, 17, 33]. Figure 3 compares the serial space and running times of SP-order with the other algorithms. As can be seen from the table, SP-order attains asymptotic optimality.

We also present a parallel SP-maintenence algorithm which is designed to run with a Cilk-like work-stealing scheduler [12, 21]. Our ***SP-hybrid*** algorithm consists of two tiers: a ***global tier*** based on our SP-order algorithm, and a ***local tier*** based on the Nondeterminator's SP-bags algorithm. Suppose that a fork-join program has n threads, T_1 work, and a critical-path length of T_∞. Whereas the Cilk scheduler executes a computation with work T_1 and critical-path length T_∞ in asymptotically optimal $T_P = O(T_1/P + T_\infty)$ expected time on P processors, SP-hybrid executes the computation in $O((T_1/P + PT_\infty)\lg n)$ time on P processors while maintaining SP relationships. Thus, whereas the underlying computation achieves linear speedup when $P = O(T_1/T_\infty)$, SP-hybrid achieves linear speed-up when $P = O(\sqrt{T_1/T_\infty})$, but the work is increased by a factor of $O(\lg n)$.

The remainder of this paper is organized as follows. We present the SP-order algorithm in Section 2. Section 3 presents an overview of the parallel SP-hybrid algorithm. Section 4 describes the organization of SP-hybrid's global tier in more detail, and Section 5 describes the local tier. Section 6 provides a proof of correctness, and Section 7 analyzes the performance of SP-hybrid. Finally, Section 8 reviews related work, and Section 9 offers some concluding remarks.

2 The SP-order algorithm

This section presents the serial SP-order algorithm. We begin by discussing how an SP parse tree, provided as input to SP-order, is created. We then review the concept of an English-Hebrew ordering [27], showing that two linear orders are sufficient to capture SP relationships. We show how to maintain these linear orders on the fly using order-maintenance data structures [10, 15, 17, 33]. Finally, we give the SP-order algorithm itself. We show that if a fork-join multithreaded program has a parse tree with n leaves, then the total time for on-the-fly construction of the SP-order data structure is $O(n)$ and each SP query takes $O(1)$ time. Thus, any fork-join program running in T_1 time on a single processor can be checked on the fly for determinacy races in $O(T_1)$ time.

The input to SP-order

SP-order takes as input a fork-join multithreaded program expressed as an SP parse tree. In a real implementation, such as a race detector, the parse tree unfolds dynamically and implicitly as the multithreaded program executes, and the particular unfolding depends on how the program is scheduled on the multiprocessor computer. For ease of presentation, however, we assume that the program's SP parse tree unfolds according to a left-to-right tree walk. During this tree walk, SP-order maintains the SP relationships "on the fly" in the sense that it can immediately respond to SP queries between any two executed threads. At the end of the section, we relax the assumption of left-to-right unfolding, at which point it becomes apparent that no matter how the parse tree unfolds, SP-order can maintain SP relationships on the fly.

English and Hebrew orderings

SP-order uses two total orders to determine whether threads are logically parallel, an ***English order*** and a ***Hebrew order***. In the English order, the nodes in the *left* subtree of a P-node precede those in the *right* subtree of the P-node. In the Hebrew order, the order is reversed: the nodes in the *right* subtree of a P-node precede those in the *left*. In both orders, the nodes in the left subtree of an S-node precede those in the right subtree of the S-node.

Figure 4 shows English and Hebrew orderings for the threads in the parse tree from Figure 2. Notice that if u_i belongs to the left subtree of an S-node and u_j belongs to the right subtree of the same S-node, then we have $E[u_i] < E[u_j]$ and $H[u_i] < H[u_j]$. In contrast, if u_i belongs to the left subtree of a P-node and u_j belongs to the right subtree of the same P-node, then $E[u_i] < E[u_j]$ and $H[u_i] > H[u_j]$.

The English and Hebrew orderings capture the SP relationships in the parse tree. Specifically, if one thread u_i precedes another thread u_j in both orders, then thread $u_i \prec u_j$ in the parse tree (or multithreaded dag). If u_i precedes u_j in one order but u_i follows u_j in the other, then $u_i \parallel u_j$. For example, in Figure 4, we have $u_1 \prec u_4$, because $1 = E[u_1] < E[u_4] = 4$ and $5 = H[u_1] < H[u_4] = 8$. Similarly, we can deduce that $u_1 \parallel u_6$, because $1 = E[u_1] < E[u_6] = 6$ and $5 = H[u_1] > H[u_6] = 3$. The following lemma shows that this property always holds.

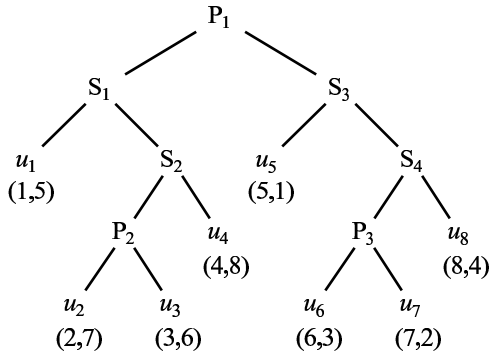

Figure 4: An English ordering E and a Hebrew ordering H for the threads in the parse tree from Figure 2. Under each thread u is an ordered pair $(E[u], H[u])$ giving its index in each of the two orders.

Lemma 1. *Let E be an English ordering of the threads of an SP-parse tree, and let H be a Hebrew ordering. Then, for any two threads u_i and u_j in the parse tree, we have $u_i \prec u_j$ in the parse tree if and only if $E[u_i] < E[u_j]$ and $H[u_i] < H[u_j]$.*

Proof. ($\Rightarrow$) Suppose that $u_i \prec u_j$, and let $X = \text{lca}(u_i, u_j)$. Then, X is an S-node in the parse tree, the thread u_1 resides in X's left subtree, and u_j resides in X's right subtree. In both orders, the threads in the X's left subtree precede those in X's right subtree, and hence, we have $E[u_i] < E[u_j]$ and $H[u_i] < H[u_j]$.

($\Leftarrow$) Suppose that $E[u_i] < E[u_j]$ and $H[u_i] < H[u_j]$, and let $X = \text{lca}(u_i, u_j)$. Since we have $E[u_i] < E[u_j]$, thread u_1 must appear in X's left subtree, and u_2 must appear in X's right subtree. By definition of a Hebrew ordering, X must be an S-node, and hence $u_i \prec u_j$. □

We can restate Lemma 1 as follows.

Corollary 2. *Let E be an English ordering of the threads of an SP-parse tree, and let H be a Hebrew ordering. Then, for any two threads u_i and u_j in the parse tree with $E[u_i] < E[u_j]$, we have $u_i \parallel u_j$ if and only if $H[u_i] > H[u_j]$.*

Labeling a static SP parse tree with an English-Hebrew ordering is easy enough. To compute the English ordering, perform a depth-first traversal visiting left children of both P-nodes and S-nodes before visiting right children (an ***English walk***). Assign label i to the ith thread visited. To compute the Hebrew ordering, perform a depth-first traversal visiting right children of P-nodes before visiting left children but left children of S-nodes before visiting right children (a ***Hebrew walk***). Assign labels to threads as before.

In race-detection applications, one must generate "on-the-fly" orderings as the parse tree unfolds. If the parse tree unfolds according to an English walk, then computing an English ordering is easy. Unfortunately, computing a Hebrew ordering on the fly during an English walk is problematic. In the Hebrew ordering the label of a thread in the left subtree of a P-node depends on the number of threads in the right subtree. In an English walk, however, this number is unknown until the right subtree has unfolded.

Nudler and Rudolph [27], who introduced English-Hebrew labeling for race detection, addressed this problem by using large thread labels. In particular, the number of bits in a label in their scheme can grow linearly in the number of P-nodes in the SP parse tree. Although they gave a heuristic for reducing the size of labels, manipulating large labels is the performance bottleneck in their algorithm.

The SP-order data structure

Our solution is to employ order-maintenance data structures [10, 15, 17, 33] to maintain the English and Hebrew orders rather than using

the static labels described above. In order-maintenance data structures, the labels inducing the order change during the execution of the program. An order-maintenance data structure is an abstract data type that supports the following operations:

- OM-PRECEDES(L, X, Y): Return TRUE if X precedes Y in the ordering L. Both X and Y must already exist in the ordering L.
- OM-INSERT$(L, X, Y_1, Y_2, \ldots, Y_k)$: In the ordering L, insert new elements $Y_1, Y_2, \ldots, Y_k$, in that order, immediately after the existing element X.

The OM-PRECEDES operation can be supported in $O(1)$ worst-case time. The OM-INSERT operation can be inserted in $O(1)$ worst-case time for each node inserted.

The SP-order data structure consists of two order-maintenance data structures to maintain English and Hebrew orderings.[2] With the SP-order data structure, the implementation of SP-order is remarkably simple.

Pseudocode for SP-order

Figure 5 gives C-like pseudocode for SP-order. The code performs a left-to-right tree walk of the input SP parse tree, executing threads on the fly as the parse tree unfolds. In lines 1–3, the code handles a leaf in the SP parse tree. In a race-detection application, queries of the two order-maintenance data structures are performed in the EXECUTETHREAD function, which represents the computation of the program under test. Typically, a determinacy-race detector performs $O(1)$ queries for each memory access of the program under test.

As the tree walk encounters each internal node of the SP parse tree, it performs OM-INSERT operations into the English and Hebrew orderings. In line 4, we update the English ordering for the children of the node X and insert X's (left and right) children after X with X's left child appearing first. Similarly, we update the Hebrew ordering in lines 5–7. For the Hebrew ordering, we insert X's children in different orders depending on whether X is an S-node or a P-node. If X is an S-node, handled in line 6, we insert X's left child and then X's right child after X in the Hebrew order. If X is a P-node, on the other hand, X's left child follows X's right child. In lines 8–9, the code continues to perform the left-to-right tree walk. We determine whether X precedes Y, shown in lines 10–11, by querying the two order-maintenance structures using the order-maintenance query OM-PRECEDES.

The following lemma demonstrates that SP-ORDER produces English and Hebrew orderings correctly.

Lemma 3. *At any point during the execution of* SP-ORDER *on an SP parse tree, the order-maintenance data structures Eng and Heb maintain English and Hebrew, respectively, orderings of the nodes of the parse tree that have been visited thus far.*

Proof. Consider an internal node Y in the SP parse tree, and consider first the Eng data structure. We must prove that all the nodes in Y's left subtree precede all the nodes in Y's right subtree in the Eng ordering. We do so by showing that this property is maintained as an invariant during the execution of the code. The only place that the Eng data structure is modified is in line 4. Suppose that the invariant is maintained before SP-ORDER is invoked on a node X. There are four cases:

1. $X = Y$: Trivial.
2. X resides in the left subtree of Y: We already assume that X precedes all the nodes in Y's right subtree. In line 4, X's children are inserted immediately after X in Eng. Hence, $left[X]$ and $right[X]$ also precede all the nodes in Y's right subtree.
3. X resides in the right subtree of Y: The same argument applies as in Case 2.
4. X lies outside of the subtree rooted at Y: Inserting X's children anywhere in the data structure cannot affect the invariant.

The argument for the Heb data structure is analogous, except that one must consider the arguments for Y being a P-node or S-node separately. □

[2] In fact, the English ordering can be maintained implicitly during a left-to-right tree walk. For conceptual simplicity, however, this paper uses order-maintenance data structures for both orderings.

```
SP-ORDER(X)
1  if ISLEAF(X)
2     then EXECUTETHREAD(X)
3          return

   ▷ X is an internal node
4  OM-INSERT(Eng, X, left[X], right[X])

5  if ISSNODE(X)
6     then OM-INSERT(Heb, X, left[X], right[X])
7     else OM-INSERT(Heb, X, right[X], left[X])

8  SP-ORDER(left[X])
9  SP-ORDER(right[X])

SP-PRECEDES(X, Y)
10 if OM-PRECEDES(Eng, X, Y) and
           OM-PRECEDES(Heb, X, Y)
11    then return TRUE
12 return FALSE
```

Figure 5: The SP-order algorithm written in C-like pseudocode. The SP-ORDER procedure maintains the relationships between thread nodes in an SP parse tree which can be queried using the SP-PRECEDES procedure. An internal node X in the parse tree has a left child, $left[X]$, and a right child, $right[X]$. Whether a node is an S-node or a P-node can be queried with ISSNODE. Whether the node is a leaf can be queried with ISLEAF. The English and Hebrew orderings being constructed are represented by the order-maintenance data structures Eng and Heb, respectively.

The next theorem shows that SP-PRECEDES works correctly.

Theorem 4. *Consider any point during the execution of the* SP-ORDER *procedure on an SP parse tree, and let u_i and u_j be two threads that have already been visited. Then, the procedure* SP-PRECEDES(u_i, u_j) *correctly returns* TRUE *if $u_i \prec u_j$ and* FALSE *otherwise.*

Proof. The SP-ORDER procedure inserts a node X into the Eng and Heb linear orders when it visits X's parent and before executing SP-ORDER(X). Thus, any thread is already in the order-maintenance data structures by the time it is visited. Combining Lemma 1 and Lemma 3 completes the proof. □

We now analyze the running time of the SP-order algorithm.

Theorem 5. *Consider a fork-join multithreaded program having a parse tree with n leaves. Then, the total time for on-the-fly construction of the SP-order data structure is $O(n)$.*

Proof. A parse tree with n leaves has at most $O(n)$ nodes, causing $O(n)$ calls to OM-INSERT. Since each of these operations can be supported in $O(1)$ amortized time, the theorem follows. □

The following corollary explains that SP-order can be used to make an efficient, on-the-fly race detector.

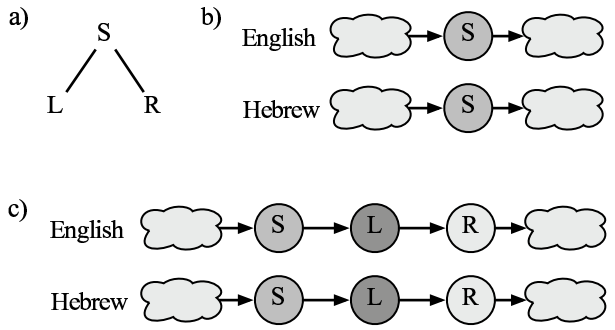

Figure 6: An illustration of how SP-order operates at an S-node. (a) A simple parse tree with an S-node S and two children L and R. (b) The order structures before traversing to S. The clouds represent the rest of the order structure, which does not change when traversing to S. (c) The result of the inserts after traversing to S. The left child L and then the right child R are inserted after S in both lists.

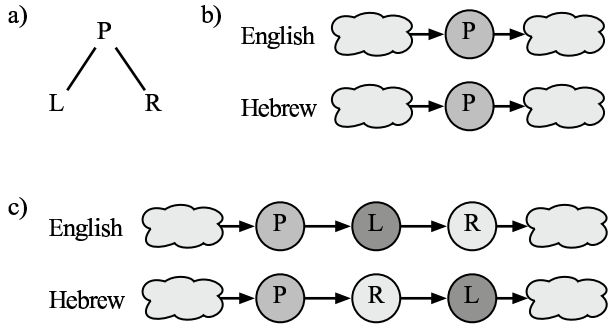

Figure 7: An illustration of how SP-order operates at a P-node. (a) A simple parse tree with an P-node P and two children L and R. (b) The order structures before traversing to P. The clouds are the rest of the order structure, which does not change when traversing to P. (c) The result of the inserts after traversing to P. The left child L then the right child R are inserted after P in the English order, and R then L are inserted after P in the Hebrew order.

Corollary 6. *Consider a fork-join multithreaded program with running time T_1 on a single processor. Then, a determinacy-race detector using SP-order runs in $O(T_1)$ time.* □

To conclude this section, we observe that SP-order can be made to work on the fly no matter how the input SP parse tree unfolds. Not only can lines 8–9 of Figure 5 be executed in either order, the basic recursive call could be executed on nodes in any order that respects the parent-child and SP relationships. For example, one could unfold the parse tree in essentially breadth-first fashion at P-nodes as long as the left subtree of an S-node is fully expanded before its right subtree is processed. An examination of the proof of Lemma 3 shows why we have this flexibility. The invariant in the proof considers only a node and its children. If we expand any single node, its children are inserted into the order-maintenance data structures in the proper place independent of what other nodes have been expanded.

3 The SP-hybrid algorithm

This section describes the structure of the SP-hybrid parallel SP-maintenance algorithm. We begin by discussing how an SP parse tree is provided as input to SP-hybrid and explaining some of the properties of Cilk that SP-hybrid exploits. We then describe the two-tier structure of the algorithm, which combines elements of SP-order from Section 2 and SP-bags from [20]. We investigate the synchronization issues that must be faced in order to parallelize SP-order and why a naive parallelization does not yield good bounds. We then overview SP-hybrid itself and present pseudocode for its implementation.

SP-hybrid's input and Cilk

Like the SP-order algorithm, the SP-hybrid algorithm accepts as input a fork-join multithreaded program expressed as an SP parse tree. The algorithm SP-hybrid provides weaker query semantics than the serial SP-order algorithm; these semantics are exactly what is required for on-the-fly determinacy-race detection. Whereas SP-order allows queries of any two threads that have been unfolded in the parse tree, SP-hybrid requires that one of the threads be a currently executing thread. For a fork-join program with n threads, T_1 work, and a critical path of length T_∞, the parallel SP-hybrid algorithm can be made to run (in Cilk) in $O((T_1/P + PT_\infty)\lg n)$ expected time.

Although SP-hybrid provides these performance bounds for any fork-join program, it can only operate "on the fly" for programs whose parse trees unfold in a Cilk-like manner. Specifically, SP-hybrid is described and analyzed as a Cilk program, and as such, it takes advantage of two properties of the Cilk scheduler to ensure efficient execution. First, any single processor unfolds the parse tree left-to-right. Second, it exploits the properties of Cilk's "work-stealing" scheduler, both for correctness and efficiency. Although SP-hybrid operates correctly and efficiently on the *a posteriori* SP parse tree for any fork-join program, it only operates "on-the-fly" when the parse tree unfolds similar to a Cilk computation.

Cilk employs a "work-stealing" scheduler, which executes any multithreaded computation having work T_1 and critical-path length T_∞ in $O(T_1/P + T_\infty)$ expected time on P processors, which is asymptotically optimal. The idea behind work stealing is that when a processor runs out of its own work to do, it "steals" work from another processor. Thus, the steals that occur during a Cilk computation break the computation, and hence the computation's SP parse tree, into a set of "traces," where each trace consists of a set of threads all executed by the same processor. These traces have additional structure imposed by Cilk's scheduler. Specifically, whenever a thief processor steals work from a victim processor, the work stolen corresponds to the right subtree of the P-node that is highest in the SP-parse tree walked by the victim. Cilk's scheduler provides an upper bound of $O(PT_\infty)$ steals with high probability.

A naive parallelization of SP-order

A straightforward way to parallelize the SP-order algorithm is to share the SP-order data structure among the processors that are executing the input fork-join program. The problem that arises, however, is that processors may interfere with each other as they modify the data structure, and thus some method of synchronization must be employed to provide mutual exclusion.

A common way to handle mutual exclusion is through the use of locks. For example, suppose that each processor obtains a global lock prior to every OM-INSERT or OM-PRECEDES operation on the shared SP-order data structure, releasing the lock when the operation is complete. Although this parallel version of SP-order is correct, the locking can introduce significant performance penalties.

Consider a parallel execution of this naive parallel SP-order algorithm on P processors. During a single operation by a processor on the shared SP-order data structure, all $P - 1$ other processors may stall while waiting for the lock required to perform their own operations. Let us assume, as is reasonable, that no processor waits on a lock unless another processor owns the lock. Thus, if we attribute the cost of waiting for a lock to the processor that owns the

lock (rather than to the processor doing the waiting), the amortized cost of a single operation could be as large as $\Theta(P)$. Since as many as $\Theta(T_1)$ operations could occur during an execution of a fork-join program with work T_1, the ***apparent work*** — the real work plus any time spent by processors waiting for locks — could expand to $\Theta(PT_1)$, thereby negating any benefits of P-way parallelism.

Of course, this scenario provides a worst-case example, and common programs may not realize such a pessimistic bound. Nevertheless, locking can significantly inhibit the scalability of a parallel algorithm, and we would like provable guarantees on scalability.

Overview of SP-hybrid

The SP-hybrid algorithm uses a two-tiered hierarchy with a global tier and a local tier in order to overcome the scalability problems with lock synchronization. As SP-hybrid performs a parallel walk of the input SP parse tree, it partitions the threads into traces on the fly, where each trace consists of threads that execute on the same processor. Much as in the naive parallelization of SP-order, the global tier of SP-hybrid uses a shared SP-order algorithm to maintain the relationships between threads belonging to different traces. The local tier uses the serial SP-bags algorithm to maintain the relationships between threads belonging to the same trace.

The goal of this two-tier structure is to reduce the synchronization delays for shared data structures, that is, processors wasting their time by waiting on locks. SP-hybrid's shared global tier minimizes synchronization delays in two ways. First, a lock-free scheme is employed so that OM-PRECEDES can execute on the shared data structure without locking. Second, the number of insertions is reduced to $O(PT_\infty)$, thereby reducing the maximum apparent work for insertions to $O(P^2T_\infty)$, since at most $P-1$ processors need to wait during the work of any insertion.

For the purposes of explaining how SP-hybrid works, we maintain traces explicitly. Formally, we define a ***trace*** U to be a (dynamic) set of threads that have been executed on a single processor. The ***computation*** $\mathcal{C}$ is a dynamic collection of disjoint traces, $\mathcal{C} = \{U_1, U_2, \ldots, U_k\}$. Initially, the computation consists of a single empty trace. As the computation unfolds, each thread is inserted into a trace.

Whenever Cilk's scheduler causes a steal from a victim processor that is executing a trace U, SP-hybrid splits U into five subtraces $\langle U^{(1)}, U^{(2)}, U^{(3)}, U^{(4)}, U^{(5)}\rangle$, modifying the computation $\mathcal{C}$ as follows:

$$\mathcal{C} \leftarrow \mathcal{C} - U \cup \{U^{(1)}, U^{(2)}, U^{(3)}, U^{(4)}, U^{(5)}\} \ .$$

Consequently, if the Cilk scheduler performs s steals, $|\mathcal{C}| = 4s+1$. Since the Cilk scheduler provides a bound of $O(PT_\infty)$ steals with high probability, the expected size of $\mathcal{C}$ is $O(PT_\infty)$. The principal use of the SP-bags algorithm from [20] is that it enables efficient splitting, as will be explained in Section 5.

Details of the two tiers of SP-hybrid will be presented in Sections 4 and 5. For now, it is sufficient to understand the operations each tier supports. The global tier supports the operations OM-INSERT and OM-PRECEDES on English and Hebrew orderings. In addition, the global tier supports a OM-MULTI-INSERT operation, which inserts several items into an order-maintenance data structure. The local tier supports LOCAL-INSERT and LOCAL-PRECEDES on a local (SP-bags) data structure. It supports an operation SPLIT, which partitions the threads in a trace when a steal occurs. It also supports an operation FIND-TRACE, which returns the current trace to which a thread belongs. The implementation of all the local-tier operations must be such that many FIND-TRACE operations can execute concurrently.

Figure 8 presents the Cilk-like pseudocode for the SP-hybrid algorithm. (See [28] for a more complete presentation of the Cilk language.) As in the SP-order algorithm, SP-hybrid performs a left-to-right walk of the SP parse tree, executing threads as the parse tree unfolds. Each thread is inserted into a trace, which is local to the processor executing the thread. The structure of the trace forms the local tier of the SP-hybrid algorithm and is described further in Section 5.

SP-hybrid associates each node in the SP parse tree with a single trace by accepting a trace U as a parameter in addition to a node X, indicating that the descendant threads of X should be inserted into the trace U. When SP-HYBRID(X, U) completes, it returns the trace with which to associate the next node in the walk of the

```
SP-HYBRID(X, U)

     ▷ X is a SP-parse-tree node, and U is a trace
 1   if ISLEAF(X)
 2      then ▷ X is a thread
 3           U ← U ∪ {X}
 4           EXECUTE-THREAD(X)
 5           return U

 6   if ISSNODE(X)
 7      then ▷ X is an S-node
 8           U' ← spawn SP-HYBRID(left[X], U)
 9           sync
10           U'' ← spawn SP-HYBRID(left[X], U')
11           sync
12           return U''

     ▷ X is a P-node
13   U' ← spawn SP-HYBRID(left[X], U)
14   if SYNCHED()
15      then ▷ the recursive call on line 13 has completed
16           U'' ← spawn SP-HYBRID(left[X], U')
17           sync
18           return U''

     ▷ A steal has occurred
19   create new traces U^(1), U^(2), U^(4), and U^(5)
20   ACQUIRE(lock)
21   OM-MULTI-INSERT(Eng, U^(1), U^(2), U, U^(4), U^(5))
22   OM-MULTI-INSERT(Heb, U^(1), U^(4), U, U^(2), U^(5))
23   RELEASE(lock)
24   SPLIT(U, X, U^(1), U^(2))
25   spawn SP-HYBRID(left[X], U^(4))
26   sync
27   return U^(5)
```

Figure 8: The SP-hybrid algorithm written in Cilk-like pseudocode. SP-HYBRID accepts as arguments an SP-parse-tree node X and a trace U, and it returns a trace. The algorithm is essentially a tree walk which carries along with it a trace U into which encountered threads are inserted. The **spawn** keyword is a Cilk linguistic construct to indicate the forking of a subprocedure, and the **sync** keyword indicates a join of the procedure with all of the children it has spawned. The EXECUTE-THREAD procedure executes the thread and handles all local-tier operations. The SYNCHED procedure determines whether the current procedure is synchronized (whether a **sync** would cause the procedure to block), which indicates whether a steal has occurred. The OM-MULTI-INSERT(L, A, B, U, C, D) inserts the objects A, B, C, and D before and after U in the order-maintenance data structure L. The Eng and Heb data structures maintain the English and Hebrew orderings of traces. The SPLIT procedure uses node X to partition the existing threads in trace U into three sets, leaving one of the sets in U and placing the other two into $U^{(1)}$ and $U^{(2)}$.

```
SP-PRECEDES(X, Y)
28  U1 ← FINDTRACE(X)
29  U2 ← FINDTRACE(Y)
30  if U1 = U2
31     then return LOCAL-PRECEDES(X, Y)
32  if OM-PRECEDES(Eng, X, Y) and
          OM-PRECEDES(Heb, X, Y)
33     then return TRUE
34  return FALSE
```

Figure 9: The SP-Precedes precedure for the SP-Hybrid algorithm given in Figure 8. SP-PRECEDES accepts two threads X and Y and returns TRUE if $X \prec Y$. FINDTRACE and LOCAL-PRECEDES are local-tier operations to determine what trace a thread belongs to and the relationship between threads in the same trace, respectively.

parse tree. In particular, for an S-node X, the trace U' returned from the walk of the left subtree is passed to the walk of X's right subtree; see Lines 6–12. The same is true for P-nodes, unless a the right subtree has been stolen; see lines lines 13–18.

Lines 1–5 deal with the case where X is a leaf and therefore a thread. As in SP-ORDER, the queries to the SP-maintenance data structure occur in the EXECUTE-THREAD procedure. In our analysis in Section 7, we shall assume that the number of queries is at most the number of instructions in the thread. The thread is inserted into the provided trace U in line 3 before executing the thread in line 4. Lines 6–12 and lines 13–27 handle the cases where X is an S- or P-Node, respectively. For both P-nodes and S-nodes, The procedure walks to X's left then right subtree. For an S-node, however, the left subtree must be fully expanded before walking to the right subtree.

During the time that a P-node is being expanded, a steal may occur. Specifically, while the current processor walks the left subtree of the P-node, another processor may steal (the walking of) the right subtree. When a steal is detected (line 14—SYNCHED returns FALSE), the current trace is split into five traces—$U^{(1)}$, $U^{(2)}$, $U^{(3)}$, $U^{(4)}$, and $U^{(5)}$—with a call to the SPLIT procedure. This SPLIT procedure, and the partitioning into subtraces, is described further in Section 5. The SP-hybrid algorithm proceeds to order the traces, inserting the five new traces into the global SP-maintenance data structures. The Eng order maintains the English ordering of the traces, as follows:

$$\langle U^{(1)}, U^{(2)}, U^{(3)}, U^{(4)}, U^{(5)} \rangle \ .$$

Similarly, the Heb order maintains the Hebrew ordering of the traces:

$$\langle U^{(1)}, U^{(4)}, U^{(3)}, U^{(2)}, U^{(5)} \rangle \ .$$

If steal does not occur, we execute lines 16–18. Notice that if a steal does not occur anywhere in the subtree rooted at some node X, then we execute only lines 1–18 for the walk of this subtree. Thus, all descendant threads of X belong to the same trace, thereby satisfying the requirement that a trace be a set of threads that execute on the same processor.

The pseudocode for SP-PRECEDES is shown in Figure 9. A SP-PRECEDES query for threads u_i and u_j first examines the order of their respective traces. If the two threads belong to the same trace, the local-tier (SP-bags) data structure determines whether u_i precedes u_j. If the two threads belong to different traces, the global-tier SP-order data structure determines the order of the two traces.

4 The global tier

As introduced in Section 3, the global tier is essentially a shared SP-order data structure, and locking is used to mediate concurrent operations. This section describes the global tier in more detail. We show how to support concurrent queries without locking, leaving only insertions as requiring locking.

We focus on making OM-PRECEDES operations on the global tier run efficiently without locking, because the number of concurrent queries may be large. If we were to lock the data structure for each of Q queries, each query might be forced to wait for insertions and other queries, thereby increasing the apparent work by as much as $\Theta(QP)$ and nullifying the advantages of P-way parallelism. Thus, we lock the entire global tier when an insertion occurs, but use a lock-free implementation for the presumably more-numerous queries.

The global tier is implemented using an $O(1)$-amortized-time order-maintenance data structure such as those described in [10, 17, 33]. The data structure keeps a doubly linked list[3] of items and assigns an integer label to each inserted item. The labels are used to implement OM-PRECEDES: to compare two items in the linear order, we compare their labels. When OM-INSERT adds a new item to the dynamic set, it assigns the item a label that places the item into its proper place in the linear order.

Sometimes, however, an item must be placed between two items labeled i and $i+1$, in which case this simple scheme does not work. At this point, the data structure relabels some items so that room can be made for the new item. We refer to the dynamic relabeling that occurs during an insertion as a ***rebalance***. Depending on how "bunched up" the labels of existing items are, the algorithm may need to relabel different numbers of items during one rebalance than another. In the worst case, nearly all of the items may need to be relabeled.

When implementing a rebalance, therefore, the data structure may stay locked for an extended period of time. The goal of the lock-free implementation of OM-PRECEDES is to allow these operations to execute quickly and correctly even in the midst of rebalancing. The implementation of a rebalance for the global tier therefore maintains two properties which are not usually implemented in a serial order-maintenance data structure:

- A concurrent query can detect whether a rebalance in progress has corrupted its view of the linear order.
- Throughout the rebalance, the relative order of the items does not change.

The first of these properties is enforced by associating a ***timestamp*** with each item which is incremented during a rebalance. The second is enforced by performing the rebalance in two passes.

The algorithm actually proceeds in five passes, two of which implement the rebalance:[4]

1. Determine the range of items to rebalance.
2. Increment the timestamp of every item in the range to indicate the beginning of the rebalance.
3. Assign each item its minimum possible label in the range, starting with the smallest item and proceeding to the largest, thereby maintaining the correct linear order.
4. Increment the timestamp of every item in the range to indicate that the second pass has begun.
5. Assign the desired final label to each item, starting with the largest item and proceeding to the smallest, thereby maintaining the correct linear order.

[3] Actually, a two-level hierarchy of lists is maintained, but this detail is unnecessary to understand the basic workings of lock-free queries, and the one-level scheme we describe can be easily extended.

[4] The number of passes can be reduced, but this presentation favors clarity.

This rebalancing strategy modifies each item 4 times while guaranteeing that the correct linear ordering of items is maintained throughout.

OM-PRECEDES uses the timestamps to determine whether a rebalance is in progress. To compare items X and Y, it examines the label and timestamp of X, then of Y, then of X again, and finally of Y again. If the second readings of labels and timestamps produce the same values as the first readings, then the query attempt ***succeeds*** and the order of labels determines the order of X and Y. Otherwise, the query attempt ***fails*** and is repeatedly retried until it succeeds.

Given that queries attempts can fail, they may increase the apparent work and the apparent critical-path length of the computation. Section 7 bounds these increases.

5 The local tier

This section describes the local tier of the SP-hybrid algorithm. We show how a trace running locally on a processor can be split when a steal occurs. By using the SP-bags algorithm to implement the trace data structure, a split can be implemented in $O(1)$ time. Finally, we show that these data structures allow the series-parallel relationship between a currently running thread and any other previously executed or currently executing thread to be determined.

Splitting traces

Besides maintaining the SP relationships within a single trace, the local tier of the SP-hybrid algorithm supports the splitting of a trace into subtraces. A split of a trace U occurs when the processor executing U becomes the victim of a steal. The work stolen corresponds to the right subtree of the P-node X that is highest in the SP-parse tree walked by the victim. When a trace U is split around a P-node X, the local tier creates five subtraces:[5]

1. $U^{(1)} = \{u \in U : u \prec X\}$, the threads that precede X.
2. $U^{(2)} = \{u \in U : u \parallel X \text{ and } u \notin descendants(X)\}$, the threads parallel to X that do not belong to a subtree of X.
3. $U^{(3)} = \{u \in U : u \in descendants(left[X])\}$, the threads in X's left subtree.
4. $U^{(4)} = \{u \in U : u \in descendants(right[X])\}$, the threads in X's (stolen) right subtree. This set is initially empty.
5. $U^{(5)} = \{u \in U : X \prec u\}$, the threads that follow X. This set is also initially empty.

We call these properties the ***subtrace properties*** of U.

The SPLIT procedure from Figure 8 implements the split. Since $U^{(4)}$ and $U^{(5)}$ are initially empty, they are not provided as parameters to the SPLIT procedure in line 24 of the SP-HYBRID pseudocode from Figure 8. The set $U^{(3)}$ is simply those threads that remain in U after those from $U^{(1)}$ and $U^{(2)}$ have been split off.

Let us look at these subtraces in terms of the parse tree. Figure 10 shows the canonical Cilk parse tree[6] as taken from [20]. A Cilk procedure is composed of a series of sync blocks, which are implemented through a series of spawn statements followed by a single join. The form of a Cilk parse tree is slightly more restrictive than that of a generic fork-join program in Figure 2: at any given time, all the outstanding children of a procedure share the same join point.

Figure 11 shows the subtraces formed when a processor steals the tree walk rooted at $right[X]$. Since all the threads contained in

[5]In fact, the subtraces $U^{(2)}$ and $U^{(3)}$ can be combined, but we keep them separate to simplify the proof of correctness.

[6]Any SP parse tree can be represented as a Cilk parse tree with the same work and critical path by adding additional S- and P-nodes and empty threads.

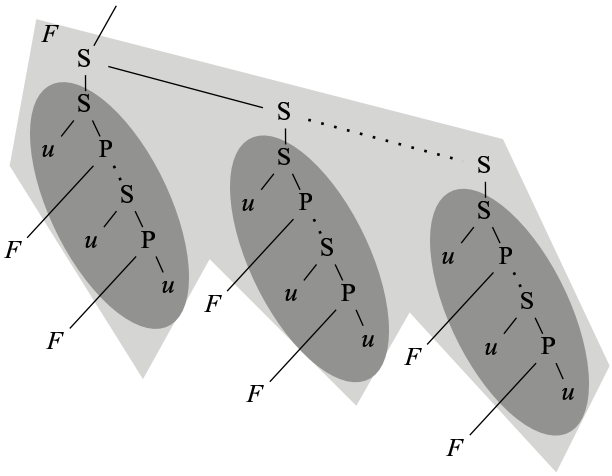

Figure 10: The canonical parse tree for a generic Cilk procedure. The notation F represents the parse tree of a spawned procedure, and u represents a thread. All the nodes in the shaded area belong to the generic procedure, while all the nodes in the ovals belong to a single sync block.

$U^{(1)}$ have been executed, no more changes to this subtrace will occur. Similarly, the threads contained in $U^{(2)}$ have already been executed. The subtrace $U^{(3)}$ is partially populated, and the processor executing the walk of U will continue to put threads into $U^{(3)}$. The subtrace $U^{(4)}$, which is initially empty, corresponds to the threads encountered during the thief processor's tree walk. The subtrace $U^{(5)}$, which is also initially empty, represents the start of the next sync block in the procedure.

When the subtraces are created, they are placed into the global tier using the concurrent SP-order algorithm. The ordering of the traces resulting from the steal in Figure 11 is shown in Figure 12. All the threads in $U^{(1)}$ precede those in $U^{(3)}$, $U^{(4)}$, and $U^{(5)}$. Similarly, all the threads (to be visited) in $U^{(5)}$ serially follow those in $U^{(1)}$, $U^{(2)}$, $U^{(3)}$, and $U^{(4)}$. Thus, we place $U^{(1)}$ first and $U^{(5)}$ last in both the English and Hebrew orders. Since any pair of threads drawn from distinct subtraces $U^{(2)}$, $U^{(3)}$, and $U^{(4)}$ operate logically in parallel, we place $U^{(2)}$, $U^{(3)}$, and $U^{(4)}$ in that order into the English ordering and $U^{(4)}$, $U^{(3)}$, and $U^{(2)}$ in that order into the Hebrew ordering.

SP-bags

The SP-bags algorithm [20] provides an efficient means for implementing the local tier using a disjoint-set data structure [14, 29]. In SP-bags, each Cilk procedure maintains two ***bags*** (sets) of threads with the following contents at any given time:[7]

- The ***S-bag*** of a procedure F contains the descendant threads of F that logically precede the currently executing thread in F. (The descendant threads of F include the threads of F.)
- The ***P-bag*** of a procedure F contains the descendant threads of child procedures of F that have returned to F and that operate logically in parallel with the currently executing thread in F.

As SP-bags walks the parse tree of the computation, it inserts threads into the bags, unions the bags, and queries as to what type of bag a procedure belongs to. SP-bags can be adapted to implement the local-tier operations LOCAL-INSERT, LOCAL-PRECEDES, FIND-TRACE, and SPLIT required by SP-hybrid. All

[7]This version of SP-bags uses bags containing threads rather than bags containing procedures, as was done in [13, 20]. This modification is straightforward to implement, and we do not dwell on the details.

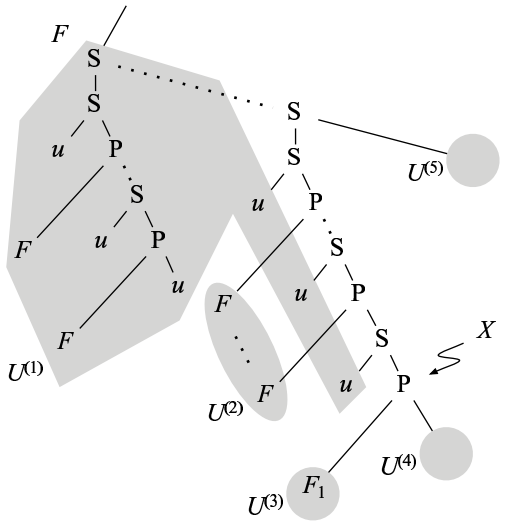

Figure 11: The split of a trace U around a P-node X in terms of a canonical Cilk parse tree. The tree walk of U is executing in $left[X]$ when the subtree rooted at $right[X]$ is stolen by a thief processor. The shaded regions contain the nodes belonging to each of the subtraces produced by the split. The two circles not enclosing any text indicate portions of the parse tree that have not yet been visited by the tree walk of U.

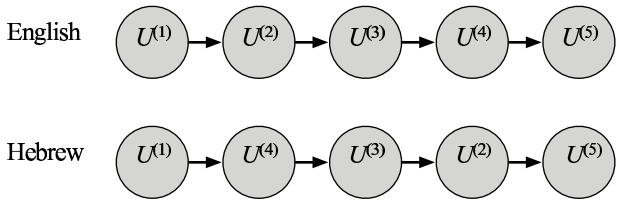

Figure 12: An ordering of the new traces resulting from a steal as shown in Figure 11. Each circle represents a trace.

these operations, except FIND-TRACE, are executed only by the single processor working on a trace. The FIND-TRACE operation, however, may be executed by any processor, and thus the implementation must operate correctly in the face of multiple FIND-TRACE operations.

The implementation of SP-bags proposed in [20] uses the classical disjoint-set data structure with "union by rank" and "path compression" heuristics [14, 29, 31]. On a single processor, this data structure allows all local-tier operations to be supported in amortized $O(\alpha(m, n))$ time, where α is Tarjan's functional inverse of Ackermann's function, m is the number of local-tier operations, and n is the number of threads. Moreover, the worst-case time for any operation is $O(\lg n)$.

The classical disjoint-set data structure does not work "out of the box" when multiple FIND-TRACE operations execute concurrently. The reason is that although these operations are queries, the path-compression heuristic modifies the data structure, potentially causing concurrent operations to interfere.[8] Consequently, our implementation of the local tier uses the disjoint-set data structure with union by rank only, which supports each operation in $O(\lg n)$ worst-case time.

The SP-bags implementation used by SP-hybrid follows that of [20], except that we must additionally support the SPLIT operation. At the time of a split, the subtraces $U^{(1)}$, $U^{(2)}$, and $U^{(3)}$ may all contain many threads. Thus, splitting them off from the trace U may take substantial work. Fortunately, SP-bags overcomes this difficulty by allowing a split to be performed in $O(1)$ time.

Consider the S- and P-bags at the time a thread in the procedure F is stolen and the five subtraces $U^{(1)}$, $U^{(2)}$, $U^{(3)}$, $U^{(4)}$, and $U^{(5)}$ are created. The S-bag of F contains exactly the threads in the subtrace $U^{(1)}$. Similarly, the P-bag of F contains exactly the threads in the subtrace $U^{(2)}$. The SP-bags data structure is such that moving these two bags to the appropriate subtraces requires only $O(1)$ pointer updates. The subtrace $U^{(3)}$ owns all the other S- and P-bags that belonged to the original trace U, and thus nothing more need be done, since $U^{(3)}$ directly inherits U's threads. The subtraces $U^{(4)}$ and $U^{(5)}$ are created with empty S- and P-bags. Thus, the split can be performed in $O(1)$ time, since only $O(1)$ bookkeeping needs to be done including updating pointers.

6 Correctness of SP-hybrid

This section proves the correctness of the SP-hybrid algorithm. We begin by showing that the traces maintained by SP-hybrid are consistent with the subtrace properties defined in Section 5. We then prove that the traces are ordered correctly to determine SP relationships. Finally, we conclude that SP-hybrid works.

Due to the way the splits work, we can no longer prove a theorem as general as Lemma 1. That is to say, we can only accurately derive the relationship between two threads if one of them is a currently executing thread.[9] Although this result is weaker than for the serial algorithm, we do not need anything stronger for a race detector. Furthermore, these are exactly the semantics provided by the lower-tier SP-bags algorithm.

The following lemma shows that when a split occurs, the subtraces are consistent with the subtraces properties given in Section 5.

Lemma 7. *Let U_i be a trace that is split around a P-node X. Then, the subtrace properties of U_i are maintained as invariants by* SP-HYBRID.

Proof. The subtrace properties of U_i hold at the time of the split around the P-node X, when the subtraces were created, by definition. If a subtrace is destroyed by splitting, the property holds for that subtrace vacuously.

Consider any thread u at the time it is inserted into some trace U. Either U is a subtrace of U_i or not. If not, then the properties hold for the subtrace U_i vacuously. Otherwise, we have five cases.

Case 1: $U = U_i{}^{(1)}$. This case cannot occur. Since $U_i{}^{(1)}$ is mentioned only in lines 19–27 of Figure 8, it follows that $U_i{}^{(1)}$ is never passed to any call of SP-HYBRID. Thus, no threads are ever inserted into $U_i{}^{(1)}$.

Case 2: $U = U_i{}^{(2)}$. Like Case 1, this case cannot occur.

Case 3: $U = U_i{}^{(3)}$. We must show that $U_i{}^{(3)} = \{u : u \in descendants(left[X])\}$. The difficulty in this case is that when the trace U_i is split, we have $U_i = U_i{}^{(3)}$, that is, U_i and $U_i{}^{(3)}$ are aliases for the same set. Thus, we must show that the invariant holds for all the already spawned instances of SP-HYBRID that took U_i as a parameter, as well as those new instances that take $U_i{}^{(3)}$ as a parameter. As it turns out, however, no new instances take $U_i{}^{(3)}$ as a parameter, because (like Cases 1 and 2) $U_i{}^{(3)}$ is neither passed to SP-HYBRID nor returned.

Thus, we are left to consider the already spawned instances of SP-HYBRID that took U_i as a parameter. One such instance is the outstanding SP-HYBRID$(left[X], U_i)$ in line 13. If $u \in$

[8] In fact, concurrent path compression does not affect the correctness of the algorithm, assuming that reads and writes execute atomically. The performance analysis become more complicated. We conjecture that a better running time can be obtained using the classical data structure.

[9] Specifically, we cannot determine the relationship between threads in $U^{(1)}$ and $U^{(2)}$, but we can determine the relationship between any other two traces.

$descendants(left[X])$, then we are done, and thus, we only need consider the spawns SP-HYBRID(Y, U_i), where Y is an ancestor of the P-node X. We use induction on the ancestors of X, starting at $Y = parent(X)$ to show that SP-HYBRID(Y, U_i) does not pass U_i to any other calls, nor does it return U_i. For the base case, we see that SP-HYBRID(X, U_i) returns ${U_i}^{(5)} \neq {U_i}^{(3)}$.

For the inductive case, consider SP-HYBRID(Y, U_i). We examine the locations in the pseudocode where this procedure can resume execution. If Y is an S-node, then this procedure can be waiting for the return from SP-HYBRID$(left[Y], U_i)$ in line 9 or SP-HYBRID$(right[Y], U_i)$ in line 11. In the first situation, our inductive hypothesis states that SP-HYBRID$(left[Y], U_i)$ does not return U_i, and hence, we neither pass U_i to the right child nor do we return it. The second situation is similar.

Instead, suppose that Y is a P-node. Since steals occur from the top of the tree, we cannot resume execution at line 16, or else SP-HYBRID$(right[Y], U_i)$ would have already been stolen. We can be only at either line 17 or line 26. If we're at line 17, our inductive assumption states that SP-HYBRID$(right[Y], U_i)$ does not return U_i, and thus we do not return U_i either. Otherwise, we are at line 26, and we return the $U^{(5)}$ resulting from some split.

Case 4: $U = {U_i}^{(4)}$. We must show that ${U_i}^{(4)} = \{u : u \in descendants(right[X])\}$. The only place where ${U_i}^{(4)}$ is passed to another SP-HYBRID call, and hence used to insert a thread, is line 25. No matter what SP-HYBRID$(right[X], {U_i}^{(4)})$ returns, SP-HYBRID(X, U_i) does not return ${U_i}^{(4)}$; it returns ${U_i}^{(5)}$. Thus, the only threads that can be inserted into ${U_i}^{(4)}$ are descendants of $right[X]$, which matches the semantics of $U^{(4)}$.

Case 5: $U = {U_i}^{(5)}$. We must show that ${U_i}^{(5)} = \{u \in U_i : X \prec u)\}$. The subtrace ${U_i}^{(5)}$ is used only in the return from SP-HYBRID(X, U_i) on line 27. As seen in lines 6–12 and lines 16–18, SP-HYBRID passes the trace returned from a left subtree to a right subtree. Thus, the only SP-HYBRID calls that have any possibility of inserting into ${U_i}^{(5)}$ are the right descendants of X's ancestors. When a split occurs (and hence when a steal occurs), by the properties of the Cilk scheduler, it occurs at the topmost P-node of a trace. Thus, the only ancestors of X with unelaborated right subtrees are S-nodes. It follows that $\mathrm{lca}(u, X)$ is an S-node, and hence $X \prec u$. □

The following lemma shows that the Eng and Heb orderings maintained by SP-hybrid are sufficient to determine the relationship between traces.

Lemma 8. *Let Eng and Heb be the English and Hebrew orderings, respectively, maintained by the global tier of SP-hybrid. Let u_j be a currently executing thread in the trace U_j, and let u_i be any thread in a different trace $U_i \neq U_j$. Then $u_i \prec u_j$ if and only if $Eng[U_i] < Eng[U_j]$ and $Heb[U_i] < Heb[U_j]$.*

Proof. The proof is by induction on the number of splits during the execution of SP-hybrid. Consider the time that a trace U is split into its five subtraces. If neither U_i nor U_j is one of the resulting subtraces $U^{(1)}, U^{(2)}, \ldots, U^{(5)}$, then the split does not affect U_i or U_j, and the lemma holds holds trivially.

Suppose that $U_i \in \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$, but $U_j \notin \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$. Then, U_i and U_j have the same relationship they did before the split, because we insert the subtraces $U^{(1)}$, $U^{(2)}$, $U^{(4)}$, and $U^{(5)}$ contiguously with $U = U^{(3)}$ in the English and Hebrew orderings. Similarly, if we have $U_i \notin \{U^{(1)}, \ldots, U^{(5)}\}$, but $U_j \in \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$, then the lemma holds symmetrically.

Thus, we are left with the situation where $U_i \in \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$, and $U_j \in \{U^{(1)}, U^{(2)}, \ldots, U^{(5)}\}$. We can ignore the case when $U_i = U_j$, because the lemma assumes that $U_i \neq U_j$, as well as the cases when $U_j \in \{U^{(1)}, U^{(2)}\}$, because u_j is a currently executing thread. We consider the remaining twelve cases in turn.

Case (1,3): $U_i = U^{(1)}$ and $U_j = U^{(3)}$. We apply Lemma 7 to conclude that $u_i \prec X$ for some P-node X and $u_j \in descendants(left[X])$, which implies that $u_i \prec u_j$. We also have that $Eng[U^{(1)}] < Eng[U^{(3)}]$ and $Heb[U^{(1)}] < Heb[U^{(3)}]$, which matches the claim.

Case (2,3): $U_i = U^{(2)}$ and $U_j = U^{(3)}$. Lemma 7 allows us to conclude that $u_i \in \{u \in U : u \parallel X \text{ and } u \notin descendants(X)\}$ for some P-node X and that $u_j \in descendants(left[X])$, which means that $u_i \parallel u_j$. We also have that $Eng[U^{(2)}] < Eng[U^{(3)}]$ and $Heb[U^{(2)}] > Heb[U^{(3)}]$, which matches the claim.

The other ten cases are similar to these two. □

We are now ready to prove that SP-hybrid returns the correct result for an SP-PRECEDES operation run on a currently executing thread and any other thread.

Theorem 9. *Consider any point during the execution of* SP-HYBRID *on an SP parse tree. Let u_i be a thread that has been visited, and let u_j be a thread that is currently executing. Then, the procedure* SP-PRECEDES(u_i, u_j) *correctly returns* TRUE *if $u_i \prec u_j$ and* FALSE *otherwise.*

Proof. The SP-HYBRID procedure inserts a thread u into a trace U before executing u, and therefore when a thread executes, it belongs to some trace. Furthermore, the English and Hebrew orderings Eng and Heb, respectively, contain all traces that contain any threads. If u_i and u_j belong to the same trace, then SP-PRECEDES returns the correct result as the result of a query on the local tier. If u_i and u_j belong to different traces, then Lemma 8 shows that the correct result is returned. □

7 Performance analysis

This section analyzes the SP-hybrid algorithm run on a fork-join program. Suppose that the program has n threads, T_1 work, and a critical-path length of T_∞. When executed on P processors using the Cilk scheduler, we prove that SP-hybrid runs in $O((T_1/P + PT_\infty) \lg n)$ expected time.

Theorem 10. *Suppose that a fork-join program has n threads, T_1 work, and a critical-path length of T_∞. When executed on P processors using the Cilk scheduler, SP-hybrid runs in $O((T_1/P + PT_\infty) \lg n)$ expected time.*

Proof. We use an accounting argument similar to [12], except with seven buckets, instead of three. Each bucket corresponds to a type of task that a processor can be doing during a step of the algorithm. For each time step, each processor places one dollar in exactly one bucket. If the execution takes time T_P, then at the end the total number of dollars in all of the buckets is PT_P. Thus, if we sum up all the dollars in all the buckets and divide by P, we obtain the running time.

The analysis depends on the number s of successful steals during the execution of the SP-hybrid algorithm. We shall show that the expected value of s is $O(PT_\infty \lg n)$. The seven buckets are as follows:

$\boldsymbol{B_1}$: The work of the original computation excluding costs added by SP-hybrid. We have that $|B_1| = O(T_1)$, because a processor places one dollar in the work bucket whenever it performs work on the input program.

$\boldsymbol{B_2}$: The work for global-tier insertions, including the cost for splits. SP-hybrid performs an OM-INSERT operation, serially, for each steal. The amortized time required to perform s operations in the order-maintenance data structure is $O(s)$. Thus, $|B_2| = O(s)$.

$\boldsymbol{B_3}$: All the other SP-maintenance operations not included in B_2. This work is dominated by the local-tier SP-bags operations. Because there are $O(1)$ SP-bags operations for each instruction in the

computation and each SP-bags operation costs $O(\lg n)$ time, we have $|B_3| = O(T_1 \lg n)$.

$\boldsymbol{B_4}$: The work wasted on synchronization delays waiting for the global lock on global-tier OM-INSERT operations. When one processor holds the lock, at most $O(P)$ processors can be waiting. Since $O(1)$ insertions occurs for each steal, we have $|B_4| = O(Ps)$.

$\boldsymbol{B_5}$: The work wasted on failed and retried global-tier queries. Since a single insertion into the order-maintenance structure can cause at most $O(1)$ queries to fail on each processor and the number of insertions is $O(s)$, we conclude that $|B_4| = O(Ps)$.

$\boldsymbol{B_6}$: Steal attempts while the global lock is not held by any processors. We use the potential argument from [9] to argue that $|B_6| = O(PT_\infty \lg n)$, but with one slight variation. We do not present the full argument here, because most of it is identical, but we do highlight the difference. The crux of their argument is that whenever a thief processor tries to steal from a victim processor, the victim loses a constant factor of its potential. In our variation, imagine blowing up each instruction in the original computation by a factor of $r = O(\lg n)$ to account for the worst-case bound on the disjoint-set data structure, thereby producing a new computation with total work $O(T_1 \lg n)$ and critical path $O(T_\infty r) = O(T_\infty \lg n)$. In this new computation, a processor may "accelerate" and execute up to r steps, corresponding to when the bookkeeping takes less than r time, but that only decreases the potential even more. The same argument bounding the number of steals still applies, but to a computation with critical path $T_\infty r$ rather than T_∞. Thus, the expected number of steals is $O(PT_\infty r) = O(PT_\infty \lg n)$.

$\boldsymbol{B_7}$: Steal attempts while the global lock is held by some processor. The global lock is held for $O(s)$ time in total, and in the worst case, all processors try to steal during this time. Thus, we have $|B_7| = O(Ps)$.

To conclude the proof, observe that $s \leq |B_6|$, because the number of successful steals is less than the number of steal attempts. Summing up all the buckets yields $O((T_1 + P^2 T_\infty) \lg n)$ expected dollars at the end of the computation, and hence, dividing by P, we obtain an expected running time of $O((T_1/P + PT_\infty) \lg n)$. In fact, this bound holds with high probability. □

For race-detection applications, this running time can be reduced to $O((T_1/P + PT_\infty) \lg(\min\{v, n\})$, where v is the number of shared-memory locations used by the program.

We suspect that the running time of SP-hybrid can be can be reduced to $O((T_1/P)\alpha(T_1, n) + PT_\infty \lg n)$, where α is Tarjan's functional inverse of Ackermann's function. The idea is to use the classical disjoint-set data structure with both union-by-rank and path-compression heuristics. Since union operations are only performed on a processor's own local-tier data structure, the only concurrency issues arise with path compressions, but these can be performed safely using the lock-free "compare-and-swap" primitive[10] [6]. This implementation achieves the same bounds as those in Theorem 10, but we conjecture that the coefficient of T_1/P can be reduced to $\alpha(T_1, n)$, which it achieves when $P = 1$. In addition, this implementation would seem to achieve close to this bound in practice, because processors are unlikely to contend as strongly as the worst-case bounds suggest.

8 Related work

This section summarizes related work on SP-maintenance and order-maintenance data structures.

[10] In fact, path compression can be performed safely with only the assumption of atomic reads and writes, but one may need a stronger assumption about the performance model to analyze this synchronization-free strategy than one does when using compare-and-swap.

Nudler and Rudolph [27] introduced the English-Hebrew labeling scheme for SP-maintenance. Each thread is assigned two labels, similar to the labeling in this paper. They do not, however, use a centralized data structure to reassign labels. Instead, label sizes grow proportionally to the maximum concurrency of the program. Mellor-Crummey [26] proposed an "offset-span labeling" scheme, which has label lengths proportional to the maximum nesting depth of forks. Although it uses shorter label lengths than the English-Hebrew scheme, the size of offset-span labels is not bounded by a constant as it is in our scheme.

The first order-maintenance data structure was published by Dietz two decades ago [15]. It supports insertions and deletions in $O(\lg n)$ amortized time and queries in $O(1)$ time. Tarjan observed [17] that updates could be supported in $O(1)$ amortized time, and the same result was obtained independently by Tsakalidis [33]. Dietz and Sleator [17] proposed two data structures, one that supports insertions and deletions in $O(1)$ amortized time and queries in $O(1)$ worst-case time and a another that supports all operations in $O(1)$ worst-case time. Bender, Cole, Demaine, Farach-Colton, and Zito [10] gave two simplified data structures whose asymptotic performance matches the data structures from [17]. Their paper also presents an implementation study of the amortized data structure.

A special case of the order-maintenance problem is the the ***online list-labeling problem*** [7, 16, 18, 23], also called the ***file maintenance problem*** [34–37]. In online list labeling, we maintain a mapping from a dynamic set of n elements to the integers in the range from 1 to u (***tags***), such that the order of the elements matches the order of the corresponding tags. Any solution to the online list-labeling problem yields an order-maintenance data structure. The reverse is not true, however, because there exists an $\Omega(\lg n)$ lower bound on the list-labeling problem [16, 18]. In file maintenance, we require that $u = O(n)$, since this restriction corresponds to the problem of maintaining a file densely packed and defragmented on disk.

Labeling schemes have been used for other combinatorial problems such as answering least-common-ancestor queries [1, 3, 5, 24] and distance queries used for routing [2, 4, 8, 22, 25, 32]. Although these problems are reminiscent of the order-maintenance problem, most solutions focus on reducing the number of bits necessary to represent the labels in a static (offline) setting.

Anderson and Woll [6] discuss concurrent union-find operations using path compression (with path halving) and union by rank. Whereas they consider multiple finds and multiple unions occurring concurrently, however, our problem is confined to single unions and multiple finds occurring concurrently.

9 Concluding remarks

This paper has focused on provably efficient parallel algorithms for SP-maintenance. As a practical matter, our algorithms are likely to perform faster than the worst-case bounds indicate, because it is rare that every lock access sees contention proportional to the number of processors. This observation can be used in practical implementations to simplify the coding of the algorithms and yield somewhat better performance in the common case. Nevertheless, we contend that the predictability of provably efficient software gives users less-frustrating experiences. Giving up on provable performance is an engineering decision that should not be taken lightly. We also believe that provably efficient algorithms are scientifically interesting in their own right.

As we were writing this paper, we repeatedly confronted the issue of how an amortized data structure interacts with a parallel scheduler. Standard amortized analysis could be applied to analyze the work of a computation, but we could not use amortization to analyze the critical path and had to settle for worst-case

bounds. Moreover, we were surprised that we needed to reprise the elaborate work-stealing analysis from [12] (with seven buckets, no less!) in order show that SP-hybrid was efficient. Are there general techniques that can allow us to develop provably good parallel algorithms without repeatedly subjecting ourselves (and readers) to such intricate and difficult mathematical arguments?

With respect to the results themselves, we have left many technical questions unanswered. Does a linear-work parallel algorithm for SP-maintenance exist? Can parallelism closer to T_1/T_∞ be achieved? Are our bounds actually tighter than what we have been able to show? Are there better data structures for SP-maintenance?

In future work, we plan to implement the SP-order and SP-hybrid algorithms and to evaluate their performance in a race-detection tool for Cilk programs.

Acknowledgments

Thanks to Bradley Kuszmaul of MIT CSAIL for numerous helpful discussions.

References

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 547–556, 2001.

[2] S. Alstrup, P. Bille, and T. Rauhe. Labeling schemes for small distances in trees. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 689–698, 2003.

[3] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 258–264, 2002.

[4] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Direct routing on trees. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 342–349, 1998.

[5] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 947–953, 2002.

[6] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 370–380, 1991.

[7] A. Andersson and O. Petersson. Approximate indexed lists. *Journal of Algorithms*, 29:256–276, 1998.

[8] M. Arias, L. J. Cowen, and K. A. Laing. Compact roundtrip routing with topology-independent node names. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 43–52, 2003.

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

[10] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the European Syposium on Algorithms*, pages 152–164, 2002.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

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

[13] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 1998.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[15] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 122–127, May 1982.

[16] P. F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Proceedings of the Scandinavian Workshop on Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142, July 1994.

[17] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 365–372, May 1987.

[18] P. F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *Proceedings of the Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, July 1990.

[19] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 1990.

[20] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 1–11, Newport, Rhode Island, June 1997.

[21] 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.

[22] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 210–219, 2001.

[23] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In S. Even and O. Kariv, editors, *Proceedings of the Colloquium on Automata, Languages, and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, July 1981.

[24] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 954–963, 2002.

[25] M. Katz, N. A. Katz, A. Korman, and D. Peleg. Labeling schemes for flow and connectivity. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 927–936, 2002.

[26] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing*, pages 24–33, 1991.

[27] I. Nudler and L. Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[28] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*, November 2001.

[29] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[30] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, October 1979.

[31] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[32] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10, 2001.

[33] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, May 1984.

[34] D. E. Willard. Inserting and deleting records in blocked sequential files. Technical Report TM81-45193-5, Bell Laboratories, 1981.

[35] D. E. Willard. Maintaining dense sequential files in a dynamic environment. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 114–121, San Francisco, California, May 1982.

[36] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proceedings of the ACM International Conference on Management of Data*, pages 251–260, Washington, D.C., May 1986.

[37] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, April 1992.