Provably Good Race Detection That Runs in Parallel

by

Jeremy T. Fineman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2005

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 3, 2005

Certified by . . . . . .
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Provably Good Race Detection That Runs in Parallel

by

Jeremy T. Fineman

Submitted to the Department of Electrical Engineering and Computer Science
on August 3, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

A multithreaded parallel program that is intended to be deterministic may exhibit nondeterminism due to bugs called ***determinacy races***. A key capability of race detectors is to determine whether one thread executes logically in parallel with another thread or whether the threads must operate in series. This thesis presents two algorithms, one serial and one parallel, to maintain the series-parallel (***SP***) relationships "on the fly" for fork-join multithreaded programs. For a fork-join program with T_1 work and a critical-path length of T_∞, the serial ***SP-Maintenance*** algorithm runs in $O(T_1)$ time. The parallel algorithm executes in the nearly optimal $O(T_1/P + PT_\infty)$ time, when run on P processors and using an efficient scheduler.

These SP-maintenance algorithms can be incorporated into race detectors to get a provably good race detector that runs in parallel. This thesis describes an efficient parallel race detector I call Nondeterminator-3. For a fork-join program T_1 work, critical-path length T_∞, and v shared memory locations, the Nondeterminator-3 runs in $O(T_1/P + PT_\infty \lg P + \min\{(T_1 \lg P)/P, vT_\infty \lg P\})$ expected time, when run on P processors and using an efficient scheduler.

Some parts of this thesis represent joint work with Michael A. Bender, Seth Gilbert, and Charles E. Leiserson.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

I would like to thank my advisor, Charles E. Leiserson, for his guidance on this project as well as introducing me to the problem in the first place. In addition to helping to develop some of the techniques presented in this thesis, he has been an invaluable asset for improving my writing and presentational skills.

I would also like to thank Michael A. Bender who, unasked, acted as a second advisor to me during my first year here at MIT. In addition to developing many of the ideas in this thesis with me, he introduced me to several other interesting problems and people.

I would like to thank Seth Gilbert in addition to Charles and Michael for their collaboration on much of the work in this thesis.

All the other members of the SuperTech group deserve some thanks as well. Those people include Bradley C. Kuszmaul, Gideon Stupp, Kunal Agrawal, Angelina Lee, Jim Sukha, John Danaher, Yuxiong He, Tushara Karunaratna, Zardosht Kasheff, Vicky Liu, Tim Olsen, and Siddhartha Sen. Everyone here has been forced to listen to my ideas at some point, and I thank them for putting up with me. I would particularly like to thank Kunal, Angelina, Jim, John, and Tushara for letting me bounce ideas off them. Tushara also uncovered a couple of bugs in my pseudocode.

I would like to thank Ali Mohammad, Sid, Kunal, and, on rare occasions, Charles, for joining me at the gym and providing me with a way to step away from academia and clear my head for a few hours.

Last and certainly least, I'd like to thank Steve Cantin for his incessant queries as to the status of my thesis. His daily, "have you finished your thesis yet?" provided additional motivation to complete my thesis in the sense that I wanted to shut him up.

This work was support in part by NSF Grant ACI-0324974 and by the Singapore MIT Alliance. Any opinions, findings and conclusions or recommendations expressed in this thesis are those of that author and do not necessarily reflect the views of the National Science Foundation (NSF).

Contents

List of Figures

Chapter 1

Introduction

When two parallel threads access the same shared memory location, and at least one of the accesses is a write, a ***data race*** occurs. Depending on how the threads are scheduled, the accesses may occur in either order, and the program may exhibit nondeterministic behavior. This nondeterminism is often a bug in the program. These race bugs are notoriously difficult to detect through normal debugging techniques. Even if the unintended behavior can be reliably reproduced, the use of normal debugging techniques like breakpoints and print statements may alter the scheduling enough to hide the bug. Figure 1-1 shows an example of a data race.

THREAD1	THREAD2
$x \leftarrow 1$	$x \leftarrow 0$ print x

Case 1	
THREAD1	THREAD2
$x \leftarrow 1$	
	$x \leftarrow 0$
	print $x = 0$

Case 2	
THREAD1	THREAD2
	$x \leftarrow 0$
$x \leftarrow 1$	
	print $x = 1$

Figure 1-1: An example of a data race. On the left are the two threads that are executing in parallel. On the right are two possible schedulings that result in different outputs.

Fork-join programming models, such as MIT's Cilk system [14, 32, 52], allow dynamic creation of threads according to a particular structure. An "on-the-fly" race detector augments the original program to discover races as the program executes. Since determining whether the program is race free for all inputs is intractable, an on-the-fly race detector typically verifies that a program is race free for a given input. In particular, if the race de-

tector does not discover a race in an ostensibly deterministic program, then no race exists (regardless of the scheduling).

A typical on-the-fly data-race detector [17, 26, 30, 43, 50] simulates the execution of the program while maintaining various data structures for determining the existence of races. An "access history" maintains a subset of threads that access each particular memory location. Another data structure maintains the series-parallel (***SP***) 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.

This thesis shows how to maintain the series-parallel (***SP***) relationships between logical threads in a multithreaded program "on the fly." We show that for fork-join programming models, this data-structuring problem can be solved asymptotically optimally. We also give an efficient parallel solution to the problem. This thesis also combines the SP-maintenance algorithms with efficient access-history algorithms to obtain provably good race detectors.

Series-parallel parse tree

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 1-2. 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*** [30]. 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 1-3 shows the

[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.

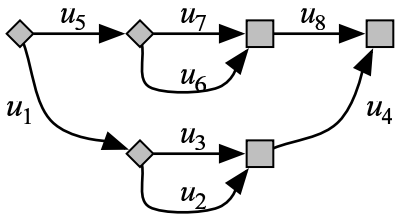

Figure 1-2: A dag representing a multithreaded computation. The edges represent threads, labeled $u_0, u_1, \ldots u_8$. The diamonds represent forks, and the squares indicate joins.

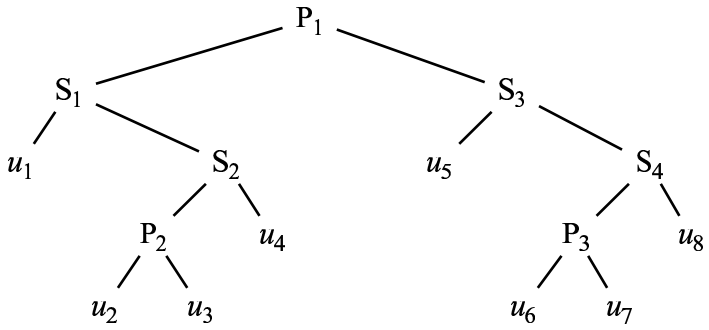

Figure 1-3: The parse tree for the computation dag shown in Figure 1-2. The leaves are the threads in the dag. The S-nodes indicate series relationships, and the P-nodes indicate parallel relationships.

parse tree corresponding to the computation dag from Figure 1-2. 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 present unless the corresponding left subtree has been fully ***elaborated***, or unfolded with all leaf threads executed. 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 1-3, 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 keeping an SP-maintenance data structure. The Nondeterminator [17, 30] race detectors use a variant of Tarjan's [54] 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 1-3, 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.

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 [50] and the ***offset-span*** scheme [43]. 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 logi-

Algorithm	Space per node	Time per	
		Thread creation	Query
English-Hebrew [50]	$\Theta(f)$	$\Theta(1)$	$\Theta(f)$
Offset-Span [43]	$\Theta(d)$	$\Theta(1)$	$\Theta(d)$
SP-Bags [30]	$\Theta(1)$	$\Theta(\alpha(v,v))$*	$\Theta(\alpha(v,v))$*
Improved SP-Bags	$\Theta(1)$	$\Theta(1)$*	$\Theta(1)$
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 1-4: Comparison of serial, SP-maintenance algorithms. An asterisk (*) indicates an amortized bound. The function α is Tarjan's functional inverse of Ackermann's function.

cally 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 thesis, I introduce a new SP-maintenance algorithm, called the ***SP-order*** algorithm, which is more efficient than Feng and Leiserson's [30] 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 [12, 21, 23, 58]. Figure 1-4 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.

I give an improvement to the SP-bags algorithms that shaves off the inverse Ackermann's $\alpha(v, v)$ factor from the running time. With this improvement, both SP-order and SP-bags are optimal SP-maintenance data structures.

I also present a parallel SP-maintenence algorithm which is designed to run with a Cilk-like work-stealing scheduler [15, 32]. The ***SP-hybrid*** algorithm consists of two tiers: a ***global tier*** based on our SP-order algorithm, and a ***local tier*** based on the improved SP-bags algorithm. Suppose that a fork-join program has 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 $O(T_1/P + PT_\infty)$ worst-case 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 speedup when $P = O(\sqrt{T_1/T_\infty})$. The parallel race detector Nondeterminator-3, which combines SP-hybrid with an efficient access-history algorithm, runs in $O(T_1/P + PT_\infty \lg P + \min\{(T_1 \lg P)/P, vT_\infty \lg P\})$ worst-case time, where v is the number of shared-memory locations being monitored.

Some results appeared earlier in a conference paper [13] jointly coauthored with Michael A. Bender, Seth Gilbert, and Charles E. Leiserson. This earlier version describes the same SP-maintenance algorithms, but this thesis describes improvements to the SP-hybrid algorithm. In particular, for a program with T_1 work, critical-path length of T_∞, and n threads, Bender et al. describe a version of SP-hybrid that runs in $O((T_1/P + PT_\infty) \lg n)$ time in expectation. This thesis improves on the result by trimming the $\lg n$ factor from the running time and by making the bound worst case.

The remainder of this paper is organized as follows. Chapter 2 presents the SP-order algorithm. Chapter 3 presents the parallel SP-hybrid algorithm. Section 3.1 gives an improvement to the SP-bags algorithm as used by SP-hybrid. Chapter 4 describes how to make SP-hybrid into a race detector, including details on the access history, resulting performance, and space usage. Finally, Chapter 5 reviews related work, and Chapter 6 offers some concluding remarks.

Chapter 2

The SP-order algorithm

This chapter presents the serial SP-order algorithm. I begin by discussing how an SP parse tree, provided as input to SP-order, is created. I then review the concept of an English-Hebrew ordering [50], showing that two linear orders are sufficient to capture SP relationships. I show how to maintain these linear orders on the fly using order-maintenance data structures [12, 21, 23, 58]. Finally, I give the SP-order algorithm itself. I 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. I conclude that any fork-join program running in T_1 time on a single processor can be checked on the fly for data 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

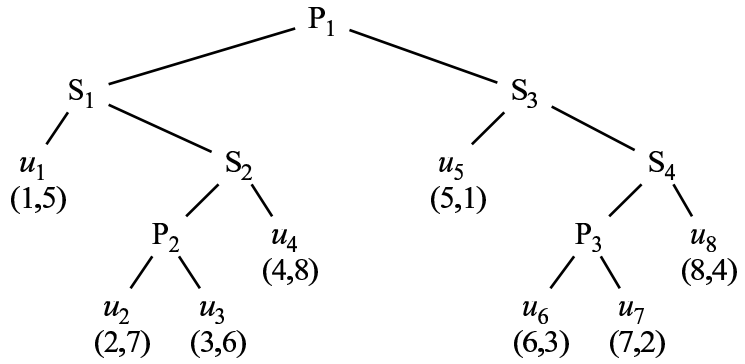

Figure 2-1: An English ordering E and a Hebrew ordering H for the threads in the parse tree from Figure 1-3. Under each thread u is an ordered pair $(E[u], H[u])$ giving its index in each of the two orders.

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 2-1 shows English and Hebrew orderings for the threads in the parse tree from Figure 1-3. 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 2-1, 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, also proved by Nudler and Rudolph [50], shows that this property always holds.

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_i 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_i must appear in X's left subtree, and u_j 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 [50], 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 [12, 21, 23, 58] to maintain the English and Hebrew orders rather than using static labels. 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 the k 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 $O(k)$ worst-case time, where k is the number of nodes being inserted (i.e., $O(1)$ time per node inserted).

The SP-order data structure consists of two order-maintenance data structures to maintain English and Hebrew orderings.[1] With the SP-order data structure, the implementation of SP-order is remarkably simple.

Pseudocode for SP-order

Figure 2-2 gives serial pseudocode for SP-order. The code traverses the input SP parse tree as a left-to-right tree walk, 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

[1] 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.

two order-maintenance data structures are performed in the EXECUTETHREAD function, which represents the computation of the program under test. Typically, a data-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. Figure 2-3 illustrates the insertions at an S-node. If X is a P-node, on the other hand, X's left child follows X's right child. Figure 2-4 illustrates these insertions. 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.

```
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 2-2: The SP-order algorithm written in serial 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 EXECUTETHREAD procedure executes the thread.

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. □

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.*

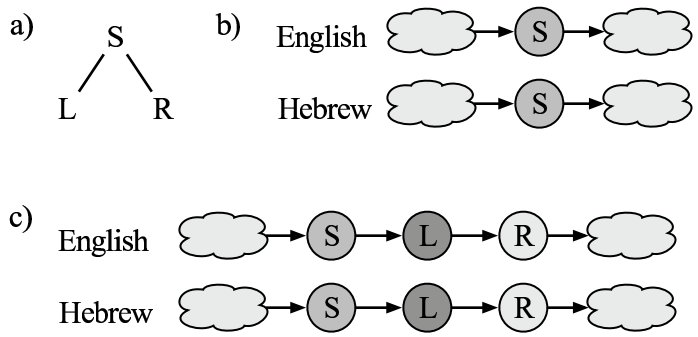

Figure 2-3: An illustration of how SP-order operates at an S-node. (a) A simple parse tree with an S-node S and two children L and R. (b) The order structures before visiting S. The clouds represent the rest of the order structure, which does not change when visiting S. (c) The result of the inserts after visiting S. The left child L and then the right child R are inserted after S in both lists.

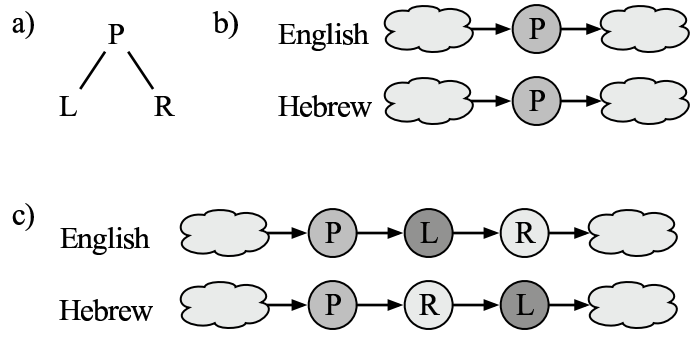

Figure 2-4: An illustration of how SP-order operates at a P-node. (a) A simple parse tree with a P-node P and two children L and R. (b) The order structures before visiting P. The clouds are the rest of the order structure, which does not change when visiting P. (c) The result of the inserts after visiting 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.

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.

Corollary 6 *Consider a fork-join multithreaded program with running time T_1 on a single processor. Then, a data-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 2-2 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.

Chapter 3

The SP-hybrid algorithm

This chapter describes the SP-hybrid algorithm, which is an SP-maintenance algorithm that runs in parallel. To support concurrent operations on a shared data structure, the algorithm uses locks. I first describe our performance model for programs that use locks. I then give lower bounds on the performance of a naive SP-order parallelization. SP-hybrid is a two-tier data structure that uses elements of SP-order from Chapter 2 and SP-bags from [30]. Section 3.1 reviews the SP-bags algorithm. I also give an improvement to the underlying data structure that allows SP-bags to run in amortized $O(1)$ time per operation. In Section 3.2 I describe the two-tier structure of the SP-hybrid. In Section 3.3, I prove the correctness of SP-hybrid. Finally, in Section 3.4, I analyze SP-hybrid's performance.

Model for parallel programs

Let us assume that SP-hybrid runs on a machine that supports the sequential consistency [40] memory model. That is to say, for any execution of the program, there is some sequential order of memory accesses that is consistent with the program order and the values observed by the program execution.

In our model, concurrent writes queue and complete in arrival order. In the case of a tie, the adversary chooses which write proceeds. Reads, however, always complete in constant time, even if there are many reads to the same location. (Thus, we do not model memory congestion in the underlying machine.) If a read is concurrent with many writes, then the

read succeeds in constant time and gets a valid value (the value before or after the write that completes on the same step).

A common way to handle mutual exclusion is through the use of locks. This technique may introduce some performance penalties. In particular, whenever a processor holds a lock, other processors may be waiting for that same lock. We call this idle time spent waiting for the lock ***waiting time***. During any step in which the lock is held by a processor, we assume the worst case—that all $P - 1$ other processors are waiting for the lock. For example, if a processor acquires the lock, performs 5 steps of real work, then releases the lock, we assume that $P - 1$ other processors were waiting for the lock during these steps, and hence these 5 steps induce a waiting time of $5(P - 1)$. Once the other processors acquire the lock, they also cause waiting time proportional to the number of steps they hold the lock. Thus, if the lock is held for L steps in total (summing across all processors), there may be $\Theta(PL)$ waiting time. To see how such a large waiting time can occur, consider a program in which P parallel threads all simultaneously try to acquire a lock, perform k steps of real work while holding the lock, and then release the lock. When the first processor acquires the lock, there are $P - 1$ other processors waiting, inducing a waiting time of $k(P-1)$. When the second processor acquires the lock, there are $P-2$ processors waiting for a waiting time of $k(P - 2)$. Summing across the waiting time introduced by all processors, we get a total waiting time of $\Theta(kP^2)$. In this example, each of the P processors holds the lock for k steps, so the lock is held for a total of $L = kP$ steps.

Locks also serialize operations. If a single lock is held for L steps in total over the entire program (again, summing across all the processors), then the running time of the program must be at least L. This bound is straightforward because none of the work performed while holding the lock can occur in parallel.

Consider a multithreaded program with work T_1 and critical-path length T_∞ in which a single lock is held for L steps across the course of the computation. By introducing $\Theta(PL)$ waiting time, we induce an ***apparent work***, the real work plus waiting time, of $T_1+\Theta(PL)$. Similarly, the serialization length L induces an ***apparent critical-path length*** of $\Theta(T_\infty+L)$.

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. In Chapter 2, we showed that the algorithm's correctness does not depend on the order of the parse tree's execution. 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.

Suppose we handle mutual exclusion 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.

Since there can be as many as $\Theta(T_1)$ SP-order operations, and each one holds the lock for an amortized constant number of steps, we hold the lock for $L = \Theta(T_1)$ steps. Thus, the apparent work becomes $\Theta(PT_1)$, and the apparent critical-path length becomes $\Theta(T_1)$. Thus, the program executes in $\Omega(T_1)$ time on P processors, which shows no asymptotic improvement over the serial SP-order algorithm.

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.

3.1 SP-bags

This section describes a variant of Feng and Leiserson's serial SP-bags algorithm [30] used by SP-hybrid. Since the SP-bags algorithm is written in terms of Cilk, I first review some Cilk terminology. Then, I describe Feng and Leiserson's algorithm. Next, I present an improvement to the underlying data structure that exploits the structure of the algorithm. This improvement results in a serial SP-maintenance algorithm that supports PRECEDES queries in worst-case constant time and INSERTs in amortized constant time.

The Cilk language [14, 32, 52] is a fork-join programming language. A Cilk program is

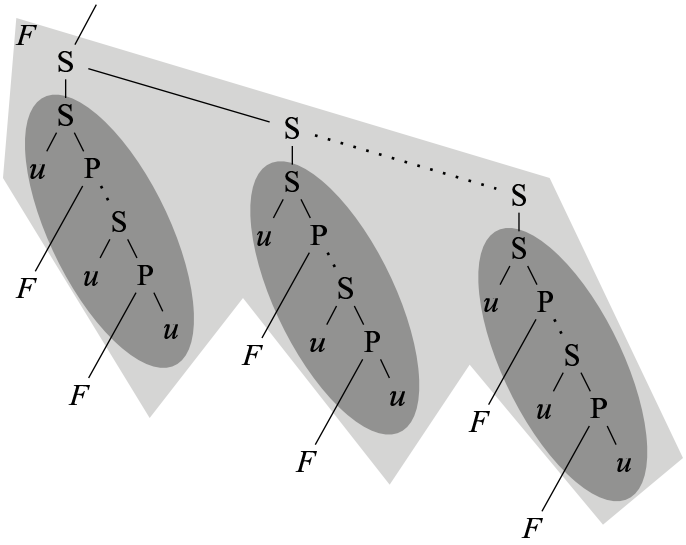

Figure 3-1: The canonical parse tree for a generic Cilk procedure. The notation F represents the parse tree of a spawned procedure, and u represents a thread. All the nodes in the shaded area belong to the generic procedure, while all the nodes in the ovals belong to a single sync block.

syntactically similar to a C program with the addition of two keywords, **spawn** and **sync**, to support parallelism. These keywords are the Cilk equivalents of fork and join, respectively. A Cilk procedure is composed of a series of sync blocks, which are implement through a series of **spawn** (fork) statements followed by a single **sync** (join). All the descendent threads of a sync block logically precede the threads of a subsequent sync block. All the descendent threads of a **spawned** procedure are logically parallel with the subsequent threads in the sync block.

Figure 3-1 shows the canonical Cilk parse tree as given by Feng and Leiserson [30]. The form of a Cilk parse tree is slightly more restrictive than that of a generic fork-join program in Figure 1-3: at any given time, all the outstanding children of a procedure share the same join point. 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.

Feng and Leiserson's SP-bags algorithm [30] uses the classical disjoint-set data structure with "union by rank" and "path compression" heuristics [20,53,55]. The data structure maintains a collection of disjoint sets and provides three operations:

1. MAKE-SET(x) creates a new set whose only member is x.
2. UNION(x, y) unites the set containing x and y.

spawn procedure F:
    $S_F \leftarrow$ MAKE-SET(F)
    $P_F \leftarrow \emptyset$

return from procedure F' to F:
    $P_F \leftarrow$ UNION$(P_F, S_{F'})$

sync in a procedure F:
    $S_F \leftarrow$ UNION(S_F, P_F)
    $P_F \leftarrow \emptyset$

Figure 3-2: The SP-bags algorithm described in terms of Cilk keywords as taken from [30]. Whenever one of three actions occurs during the serial, depth-first execution of a Cilk program, the operations in the figure are performed.

3. FIND(x) returns a representative for the set containing x.

On a single processor, this data structure allows supports m operations on n in $O(m\alpha(m, n))$ time, where α is Tarjan's functional inverse of Ackermann's function.

In SP-bags, each Cilk procedure maintains two ***bags*** (sets) of procedures with the following contents at any given time:

- The ***S-bag*** S_F of a procedure F contains the descendant procedures of F that logically precede the currently executing thread. (The descendant procedures of F include F itself.)
- The ***P-bag*** P_F of a procedure F contains the descendant procedures of F's completed children that operate logically in parallel with the currently executing thread.

As SP-bags walks the parse tree of the computation, it inserts procedures into the bags, unions the bags, and queries as to what type of bag a procedure belongs to. Figures 3-2 and 3-3 give the SP-bags algorithm in terms of the Cilk keywords [30] and the parse tree, respectively. Whenever spawning a new procedure F (entering the left subtree of a P-node), new bags are created. The bag S_F is initially set to contain F, and P_F is set to be empty. Whenever a subprocedure F' returns to its parent (going from the left subtree to the right subtree of a P-node), the contents of $S_{F'}$ are unioned into P_F, since the descendents of F' can execute in parallel with the remainder of the sync block in F. When a **sync** occurs (returning from an internal node), the bag P_F is emptied into S_F, since all of F's executed

```
SP-BAGS(X, F)
    ▷ X is an SP-parse-tree node, and F is a function.
 1  if ISLEAF(X)
 2     then ▷ X is a thread
 3          F ← F ∪ {X}
 4          EXECUTETHREAD(X, F)
 5          return

 6  if ISSNODE(X)
 7     then SP-BAGS(left[X], F)
 8          SP-BAGS(right[X], F)
 9     else ▷ X is a P-node
10          F' ← NEWPROCEDURE()
11          SP-BAGS(left[X], F')
12          P_F ← UNION(P_F, S_F')
13          SP-BAGS(right[X], F)

14  S_F ← UNION(S_F, P_F)
15  P_F ← ∅
```

Figure 3-3: The SP-bags algorithm written in serial pseudocode to operate on the canonical Cilk parsetree from Figure 3-1. SP-BAGS accepts as arguments an SP-parse-tree node X and the procedure F to which X belongs. 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. NEWPROCEDURE$[F']$ creates a new procedure object F' associated with S- and P-bags $S_{F'}$ and $P_{F'}$, initialized to $S_{F'} \leftarrow$ MAKE-SET$[F']$ and $P_{F'} \leftarrow \emptyset$, respectively.

descendents precede any future threads in F.

SP-bags supports SP-PRECEDES on two threads provided that one of the threads is the currently executing thread. Correctness of SP-bags is captured by the following lemma. Feng and Leiserson in [30] give a prove a similar lemma, and the proof is not repeated here. Note that although the algorithm given in Figure 3-3 is given in terms of a parse tree, it is correct only on parse trees that match the canonical Cilk parse tree.

Lemma 7 *Consider any point during the execution of SP-bags on a Cilk-like SP parse tree. Let u_i, belonging to a procedure F, be a thread that has been visited, and let u_j be a currently executing thread. Then, we have $u_i \parallel u_j$ if and only if u_i belongs to some P-bag. Conversely, $u_i \prec u_j$ if and only if u_i belongs to some S-bag.* □

When run on a parse-tree with T_1 work and n P-nodes (or procedures), Feng and Leiserson's SP-bags performs $\Theta(T_1)$ SP queries in the worst case. Since each query takes amortized $O(\alpha(T_1, n))$ amortized time, SP-bags runs in $O(\alpha(T_1, n)T_1)$ time.

Improving SP-bags

The SP-bags algorithm can be improved by improving the underlying disjoint-sets data structure. Gabow and Tarjan [33] describe a disjoint-sets data structure that runs in amortized-constant time per operation when the n elements being unioned are ordered *a priori* from $0, 1, \ldots, n-1$, and unions are of the form UNION$(i-1, i)$.[1] I first show that SP-bags does adhere to this structure. I then describe a slight variant of Gabow and Tarjan's data structure that has worst-case-constant time for a FIND and amortized-constant time for a UNION. Although Gabow and Tarjan [33] give most of the interesting aspects of the data structure, I present a full description for completeness, because SP-hybrid in Section 3.2 must make one additional change to the data structure, and because this data structure impacts the space usage in Section 4.3.

First, I show that SP-bags has the union structure required by Gabow and Tarjan's data structure. Consider the English ordering of procedures in the parse tree (corresponding to the left-to-right tree walk performed by SP-bags). We index these procedures $F_1, F_2, \ldots, F_n$ according to the English ordering.[2] The following lemma argues that at any point, all S- and P-bags contain contiguous procedures. Thus, all union operations effectively have the form UNION(F_{i-1}, F_i), and we can apply Gabow and Tarjan's data structure.

Lemma 8 *All* UNION*s performed by SP-bags effectively have the form* UNION(F_{i-1}, F_i).

Proof. We claim that the S- and P-bags corresponding to a procedure F_i contain the procedures $S_{F_i} = \{F_i, F_{i+1}, \ldots, F_j\}$ and $P_{F_i} = \{F_{j+1}, F_{j+2}, \ldots, F_k\}$, for some j and k with $i \leq j \leq k$. As long as this property holds across the execution of the algorithm, then we have the lemma. We prove this claim by induction on UNION operations.

[1] In fact, their algorithm is more general, but SP-bags follows this special case.

[2] Since the procedures are indexed by execution order, numbering the procedures on the fly is trivial.

As a base case, consider the S_{F_i} and P_{F_i} on creation. We have $S_{F_i} = \{F_i\}$ and $P_{F_i} = \emptyset$, which satisfies the claim. Next, consider a UNION. There are two cases.

Case 1. Suppose that a UNION occurs because of a **sync** in a procedure F_i. Then, by assumption we have $S_{F_i} = \{F_i, F_{i+1}, \ldots, F_j\}$ and $P_{F_i} = \{F_{j+1}, F_{j+2}, \ldots, F_k\}$. Thus, the result of the UNION is $S_{F_i} = \{F_i, F_{i+1}, \ldots, F_k\}$ and $P_{F_i} = \emptyset$.

Case 2. Suppose that a UNION occurs because of a **return** from a procedure $F_{i'}$ to a procedure F_i. Then, we must have $F_{i'} = F_{k+1}$ because $F_{i'}$ follows F_k in the English ordering of procedures. Moreover, since we assume that $S_{F_{i'}} = \{F_{i'}, F_{i'+1}, \ldots, F_{k'}\}$, we end with $P_{F_i} = \{F_{j+1}, F_{j+2}, \ldots, F_k, F_{k+1}, \ldots, F_{k'}\}$. □

Given Lemma 8, we could apply Gabow and Tarjan's [33] data structure as a black box to achieve a serial SP-bags algorithm that runs in amortized constant-time per operation. For SP-hybrid analysis in Section 3.4, however, where the amortization occurs is important.

The following data structure is similar to the simplified version of Gabow and Tarjan's data structure, except that the FINDs and MAKESETs are worst-case constant-time. Recall that we are given n elements $0, 1, \ldots, n-1$, and we perform unions of the form UNION$(i-1, i)$. First, assume that the n elements are known a priori. We relax this assumption later.

To achieve a linear-time union-find data structure, divide the n elements into ***microsets***. The microset boundaries are fixed and have nothing to do with the current state of the sets in the data structure. Each of these microsets contains up to w contiguous elements in the range $0, 1, \ldots, n-1$, where w is the number of bits in a machine word. Specifically, the ith microset[3] contains the elements $iw, iw+1, \ldots, (i+1)w-1$. Only the last microset can contain fewer than w elements. Thus, we have $\lceil n/w \rceil$ microsets. Note that $w = \Omega(\lg n)$, since we must be able to address each element, giving us $O(n/\lg n)$ microsets.

Each element x belongs to a microset $s = micro(x)$. Each element is assigned a microset index $index(x)$ according to its position in the microset. That is, the "smallest" element gets an index of 0, next smallest is 1, etc., up to an index of $w-1$. We call the first item (index of 0) in the microset s the ***root*** of the microset. Similarly, each microset has a list of its children as an array $node_s[0\,..\,w-1]$. In particular, $node_s[index(x)] = x$ if x belongs to microset s. We maintain all the microsets as a linked list, with pointers $prev(s)$

[3]We begin numbering microsets at 0.

and $next(s)$ to the adjacent smaller and larger, respectively, microsets.

Microsets support the following operations:

- MICROFIND(x) returns a pointer to the "smallest" element that belongs to both the same microset and the same logical set as x.
- MICROUNION(x) unions the logical sets containing x and $x-1$, where x and $x-1$ belong to the same microset.[4] That is to say, all subsequent MICROFINDs return the same result if performed on any element that belongs to both the same microset as x (and $x-1$) and the same logical set as x or $x-1$.

We describe these operations later in the section. Both of these operations are worst-case constant time.

We group these microsets into ***macrosets***. Unlike the microset structure, the macroset structure does correspond to sets represented by the data structure. Each microset s has a single macroset node $macro(s)$. In some sense, $macro(s)$ really corresponds to the root of the microset s. Macrosets support the following operations:

- MACROFIND(s) returns the "smallest" microset in the same macroset as s.
- MACROUNION(s, s') unions the macroset containing s with the macroset containing s'. These sets must be contiguous.

The implementation of macrosets is the simple linked-list implementation from [20] with the "weighted-union heuristic." On a MACROUNION, each element in the smaller set is pointed at the representative of the larger set. The set representative maintains a pointer to the smallest element in the set. This technique results in worst-case constant time MACROFINDs and a total of $O(m \lg m)$ time for all the MACROUNIONs, where m is the number of elements. In our case, there are $O(n/\lg n)$ macroset nodes, so the total running time of unions is $O((n/\lg n)\lg(n/\lg n)) = O(n)$.

The following invariant describes the representation of a logical set.

Invariant 9 *Suppose that a logical set S contains exactly the elements $x, x+1, \ldots, y$, belonging to microsets $s_i, s_{i+1}, \ldots, s_j$, where s_k denotes the kth smallest microset. Then*

[4] Since x and $x-1$ must belong to the same microset, calling a MICROUNION on the root of a microset is not supported.

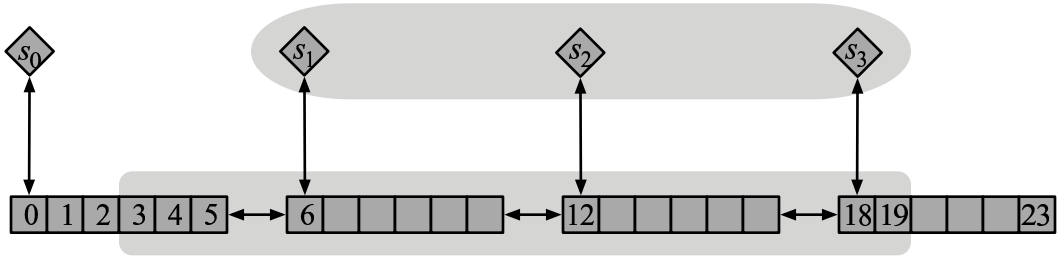

Figure 3-4: The representation of a set containing elements $3, 4, \ldots, 19$ in our disjoint-sets data structure with $n = 24$ and $w = 6$. The dark-gray rectangles on the bottom represent elements grouped into microsets. The diamonds above are the macroset nodes corresponding to each of the microsets s_0, s_1, s_2, s_3. The light-gray rounded rectangle shows the logical set being represented, containing elements $3, 4, \ldots, 19$. The light-gray oval above shows the corresponding macroset.

```
FIND(x)
1  x ← MICROFIND(x)
2  if index(x) = 0
3     then s ← MACROFIND(micro(x))
4          s ← prev(s)
5          x ← MICROFIND(node_s(w − 1))
6  return x
```

Figure 3-5: The FIND operation written in serial pseudocode. MACROFIND(s) returns the smallest microset contained in the same macroset as s. MICROFIND(x) returns the smallest element in the same logical set and microset as the element x.

there exists a macroset corresponding to the set S containing exactly $s_{i+1}, s_{i+2}, \ldots, s_j$. In other words, s_k belongs to the macroset if and only if some element in s_k belongs to S and the largest element in microset s_{k-1} also belongs S.

Figure 3-4 illustrates this representation of the set $3, 4, \ldots, 19$. The elements are contained in microsets s_0, s_1, s_2, s_3, and so the corresponding macroset contains only microsets s_1, s_2, s_3.

Figure 3-5 gives pseudocode for the FIND(x) operation. This operation returns the smallest element belonging to the same logical set as x. FIND simply searches for the smallest element in the microset containing x in line 1. If this element is the root of the microset, then it looks for the smallest microset whose root is contained in the same macroset. This value is exactly what is returned by MACROFIND in line 3 due to Invariant 9. Then, line 5 checks how far the set spans into the previous microset. For example, consider per-

```
UNION(x, y)
 1  if x > y
 2     then swap x ↔ y
 3  x' ← FIND(x)
 4  y' ← FIND(y)
 5  if x' = y'
 6     then ▷ x and y are already in the same set
 7          return

 8  if index(y') ≠ 0
 9     then MICROUNION(y')
10     else MACROUNION(next(micro(x')), micro(y'))
11  if micro(x') ≠ micro(y')
12     then s ← next(micro(y'))
13          if FIND(node_s(0)) = FIND(y')
14            then MACROUNION(micro(y'), s)
```

Figure 3-6: The UNION operation written in serial pseudocode. FIND(x), as shown in Figure 3-5, returns the smallest element in the same set as x. MICROUNION(x) unions x and $x-1$ in the same microset. MACROUNION(s, s') unions the neighboring macrosets containing s and s', respectively.

forming FIND(19) on the set shown in Figure 3-4. With the first MICROFIND in line 1, we find the element 18. This element is the root of the microset s_3. The MACROFIND in line 3 returns s_1. Invariant 9 implies that some of s_0 is contained in the set. We, therefore, perform MICROFIND(5) in line 5, which returns the answer—the node 3.

Figure 3-6 gives the implementation of a UNION(x, y) operation. Without loss of generality, y is the larger item. We first check that both items belong to different sets. Then, we look for the smallest item y' in the set containing y and perform a MICROUNION. Since $x < y$, and we union only contiguous elements, the intention must be to union y' with $y'-1$, and so a MICROUNION is correct. Lines 10–14 maintain Invariant 9—that macrosets contain all but the first microset spanned by the logical set. This update may involve up to two MACROUNIONS.

The FIND takes $O(1)$ time in the worst-case since each line has worst-case time. The UNION is dominated by the cost of a MACROUNION, which is amortized to $O(\lg n)$. I do not prove correctness here, since this data structure is quite similar to Gabow and Tar-

MICROUNION(x)
1 $s \leftarrow micro(x)$
2 $mark_s[index(x)] \leftarrow 0$.

Figure 3-7: The MICROUNION operation written in serial pseudocode.

jan's [33].

Microsets

It remains to describe the implementation of the MICROFIND and MICROUNION operations. These operations run in constant time on a random-access machine [20], where ALU operations (we use a bit-shift and modulo operation) on a machine word take constant time.

For each microset, we keep a table $mark_s[0\,..\,w-1]$. We use $mark_s[x]$ to indicate whether x belongs to the same set as any of its predecessors. If $mark_s[x] = 0$, then x and $x-1$ belong to the same set. Conversely, if $mark_s[x] = 1$, then x and $x-1$ *do not* belong to the same set. We treat $mark_s$ as an integer so that we can support arithmetic operations. In particular, if $index(x) = k$, then $mark_s[x]$ corresponds to the $(k+1)$st most significant bit in the word $mark_s$.

The operation MICROUNION(x) trivially just sets the appropriate bit in the mark table. This operation is given in Figure 3-7.

To facilitate the arithmetic operations used by MICROFINDs, it is useful to have a way to go between k and a bit string containing a 1 in only the kth bit (i.e., the integer 2^{w-k-1}). We provide two functions that work on values with $0 \leq k \leq w-1$ and $k \leq n-1$. Note that we do handle the case in which n is smaller than the size of a word.

- NUMTOSTRING(k) returns the bit string 2^{w-k-1}.
- STRINGTONUM(2^{w-k-1}) returns k.

The function NUMTOSTRING can be trivially implemented as a bit-shift operation.[5] Thus, NUMTOSTRING is a constant-time operation.

[5]Alternatively, we could precompute an array of size $\min\{w, n\}$ and fill in all the values. This construction does not require a variable-length bit shift.

```
MICROFIND(x)
1  s ← micro(x)
2  i ← NOT(mark_s − NUMTOSTRING(index(x)))
3  i ← AND(i, mark_s)
4  return node_s[STRINGTONUM(i)]
```

Figure 3-8: The MICROFIND operation written in serial pseudocode. The NOT operator performs a bitwise complement of the argument, and AND performs a bitwise conjunction of the two arguments.

The STRINGTONUM function is a bit more challenging to perform in constant time. We construct an array A of size $\Theta(\min\{w, n\})$ and provide a hash function h that indexes into the table. We precompute the $\min\{w, n\}$ values in this table such that $A[h(2^{w-k-1})] = k$. To precompute the table A, we compute $A[h(\text{NUMTOSTRING}(k))] \leftarrow k$, for all appropriate values of k. The main difficulty is picking a good hash function h that hashes onto a range of size $\Theta(\min\{w, n\})$ without any collisions for appropriate keys. One such hash function is given in Appendix A.

The operation MICROFIND(x), as shown in Figure 3-8, returns the "smallest" element (most significant bit) in the microset $s = micro(x)$ that belongs to the same set as x. This operation is equivalent to searching for the nearest 1 preceding the $(index(x) + 1)$st most significant bit in the word $mark_s$. To perform this operation, we simply subtract a 1 from the $mark_s[index(x)]$, bitwise complement the resulting word, and perform a bitwise and with $mark_s$. When subtracting, all 0s between the $(index(x) + 1)$st bit and the preceding 1 become 1s. The preceding 1 becomes a 0, and all other bits remain the same. Thus, when complementing and anding, the only bit that falls out is the preceding 1. Given this bit string, a call to STRINGTONUM gives the index of the bit. For example, consider a MICROFIND(x), where x belongs to the microset s, and $index(x) = 4$, and $mark_s$ is the bit string 11000011. Subtracting NUMTOSTRING$(index(x)) = 00001000$ from $mark_s$ yields 10111011. The bitwise complement of this string is 01000100. The subsequent AND operation of this string and $mark_s$ yields 01000000, which is a string isolating the nearest 1 preceding the 5th most significant bit.

Incremental set union

I now relax the assumption that the n items are known a priori. In other words, I describe how to support the MAKE-SET operation in worst-case $O(1)$ time. I do, however, impose the restriction that MAKE-SETs are performed in order. Since SP-bags performs MAKE-SET operations in the same order as procedures are discovered, Lemma 8 implies that this data structure does apply.

Once the number of items exceeds w, supporting worst-case $O(1)$ MAKE-SET operations is easy. We just store a pointer to the last element created. The next MAKE-SET either puts the item in the next slot in the same microset, if there is space, or creates a new microset and uses the item as the root.

Dealing with the possibility that $n < w$, however, introduces some complications. We have an array $node_s$ associated with the microset and an array A used by STRINGTONUM. If $m < w$ is the number of MAKE-SETs performed so far, we require the size of the tables to be $\Theta(m)$. To achieve this restriction, we use a standard doubling technique (as in dynamic tables [20, Section 17.4]). In the beginning, we allocate arrays of constant size (and choose an appropriate hash function for STRINGTONUM). On each subsequent MAKE-SET operation, we fill in the next table entry. Each time the number m of items doubles, we allocate new arrays of size $2m$ (and create a new hash function) and fill in the arrays again. Thus, each time the size doubles, we do $O(m)$ work. We amortize this work against the $\Theta(m)$ MAKE-SETs that occurred since the last doubling to get amortized $O(1)$ time MAKE-SETs.

We deamortize this process to get worst-case $O(1)$ time MAKE-SETs. The deamortization technique is straightforward. We keep two versions of each array. Whenever the value of m doubles, we now allocate a new array of size $4m$. Thus, at this doubling point, one array has size $2m$, and one has size $4m$. On every subsequent MAKE-SET, we fill in the appropriate table entry in the smaller table, and we fill in 2 entries in the larger table. Any queries during this time query against the smaller table, since the larger table is incomplete. At the time the next doubling occurs, since both tables contain entries for all known values, the smaller table can be discarded and replaced by a new table of size $4m$.

3.2 SP-hybrid

This section gives describes the SP-hybrid algorithm. I begin by describing how an SP parse tree is provided as input to SP-hybrid and explaining some of the properties of Cilk that SP-hybrid exploits. I then overview the two-tier structure of the algorithm which combines elements of SP-order from Chapter 2 and SP-bags from Section 3.1 and [30]. I then give 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 SP-hybrid algorithm provides weaker query semantics than the serial SP-order algorithm; these semantics are exactly what is required for on-the-fly data-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 T_1 work and a critical path of length T_∞, the parallel SP-hybrid algorithm can be made to run (in Cilk) in $O(T_1/P+PT_\infty)$ 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 [15, 32] 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, we require the following scheduling property to guarantee correctness of SP-hybrid.

Property 10

1. *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.*
2. *A processor expands the subtree it is working on in a depth-first, left-to-right manner, and steals only when its subtree has been fully expanded.*

Cilk's scheduler provides an upper bound of $O(PT_\infty)$ steals with high probability [10, 15], which bounds the number of traces created by the algorithm.

The SP-hybrid algorithm

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

waiting time 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 [30] and Section 3.1 is that it enables efficient splitting.

Details of the two tiers of SP-hybrid are presented later in the section. 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 3-9 presents the parallel pseudocode for the SP-hybrid algorithm, with details of the local tier operations omitted. (See Section 3.1 and [52] for a more complete presentation of the Cilk language.) As in the SP-order and SP-bags algorithms, SP-hybrid performs a

```
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           EXECUTETHREAD(X)
 5           return U

 6   if ISSNODE(X)
 7      then ▷ X is an S-node
 8           U′ ← SP-HYBRID(left[X], U)
 9           U″ ← SP-HYBRID(right[X], U′)
10           return U″
     ▷ X is a P-node
11   U′ ← spawn SP-HYBRID(left[X], U)
12   if SYNCHED()
13      then ▷ the recursive call on line 11 has completed
14           U″ ← spawn SP-HYBRID(right[X], U′)
15           sync
16           return U″
     ▷ A steal has occurred
17   wait until parentstolen[X] = TRUE
18   ACQUIRE(lock)
19      create new traces U^(1), U^(2), U^(4), and U^(5)
20      OM-MULTI-INSERT(Eng, U^(1), U^(2), U, U^(4), U^(5))
21      OM-MULTI-INSERT(Heb, U^(1), U^(4), U, U^(2), U^(5))
22      SPLIT(U, X, U^(1), U^(2))
23      parentstolen[left[X]] ← TRUE
24   RELEASE(lock)
25   spawn SP-HYBRID(right[X], U^(4))
26   sync
27   return U^(5)
```

Figure 3-9: The SP-hybrid algorithm written in parallel pseudocode, with the local-tier operations omitted. 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 that carries with it a trace U into which new threads are inserted. 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(u_i, u_j)
28  U_i ← FINDTRACE(u_i)
29  U_j ← FINDTRACE(u_j)
30  if U_i = U_j
31     then return LOCAL-PRECEDES(u_i, u_j)
32  if OM-PRECEDES(Eng, U_i, U_j) and
           OM-PRECEDES(Heb, U_i, U_j)
33     then return TRUE
34  return FALSE
```

Figure 3-10: The SP-Precedes precedure for the SP-Hybrid algorithm given in Figure 3-9. SP-PRECEDES accepts two threads u_i and u_j, where u_i must be a currently executing thread, and returns TRUE if $u_i \prec u_j$. 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.

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 later in the section. The full SP-hybrid algorithm can be obtained by merging Figure 3-9 with the SP-parse-tree walk performed by the local-tier algorithm.

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 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 6–10. The same is true for P-nodes, unless a the right subtree has been stolen; see lines lines 11–16.

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 EXECUTETHREAD procedure. In our analysis in Section 3.4, 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 6–10 and lines 11–27 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 12—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 later in the section. 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:

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

We use a global lock to serialize these trace constructions and insertions into the shared order-maintenance data structure. For correctness, we require that the parent procedure be stolen (and have its traces split) before the child is stolen. To enforce this condition, we introduce the field $parentstolen$ for each node in the parse tree. This value is initially set to FALSE. When a P-node is stolen, we update $parentstolen$ for the left child of the P-node. In this way, we guarantee that the trace splitting in lines 19–23 occurs only when all the ancestor procedures have been dealt with.

If a steal does not occur, we execute lines 14–16. Notice that if a steal does not occur anywhere in the subtree rooted at some node X, then we execute only lines 1–16 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 3-10. 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.

The global tier

The global tier is essentially a shared SP-order data structure, and locking is used to mediate concurrent operations. We now describe 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 [12, 23, 58]. The data structure keeps a doubly linked list[6] 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. We modify the order-maintenance data structure to contain two sets of labels—an item x

[6] 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.

has labels $label_1[x]$ and $label_2[x]$. Implementation of a rebalance maintains the following properties:

- When no rebalance is in progress, $label_1[x] = label_2[x]$ for all items x in the list, and the labels respect the total order (i.e., $label_i[x] < label_i[y]$ if and only if $x \prec y$).
- At any instant in time (during a rebalance), at least one set of labels is consistent with the total order.
- A concurrent query can detect whether a rebalance in progress has corrupted its view of the linear order.

We use a counter (which starts at 1) to support the third property. When the counter is odd, the set of $label_1$ respects the total order. When the counter is even, the set of $label_2$ is valid. The algorithm actually proceeds in five phases, two of which implement the normal rebalance:

1. Determine the range of items to rebalance.
2. Assign the desired label to each item's $label_2$.
3. Increment the counter indicating that a concurrent query should read the $label_2$'s.
4. Assign the desired label to each item's $label_1$.
5. Increment the global counter indicating that the rebalance has completed and that a concurrent query should read the $label_1$.

This rebalancing strategy modifies each item twice while guaranteeing that a concurrent read can get a consistent view of the linear order.

OM-PRECEDES query checks the counter to determine whether a rebalance is in progress. To compare items X and Y, it first determines the parity of the counter, then examines the appropriate labels of X and Y, and finally checks the counter again. If the counter has not changed between readings, then the query attempt ***succeeds***, and the order of labels determines the order of X. 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 3.4 bounds these increases.

The local tier

We now describe 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.

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:[7]

1. $U^{(1)} = \{u \in U : u \prec X\}$, the threads that precede X.
2. $U^{(2)} = \{u \in U : u \parallel X \text{ and } u \notin \mathit{descendants}(X)\}$, the threads parallel to X that do not belong to a subtree of X.
3. $U^{(3)} = \{u \in U : u \in \mathit{descendants}(\mathit{left}[X])\}$, the threads in X's left subtree.
4. $U^{(4)} = \{u \in U : u \in \mathit{descendants}(\mathit{right}[X])\}$, 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 the properties of these sets the ***subtrace properties*** of U.

The SPLIT procedure from Figure 3-9 implements the split. Since $U^{(4)}$ and $U^{(5)}$ are initially empty, they are not provided as parameters to the SPLIT procedure in line 22 of the SP-HYBRID pseudocode from Figure 3-9. 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 3-11 shows the subtraces formed when a processor steals the tree walk rooted at $\mathit{right}[X]$. Since all the threads contained in $U^{(1)}$ 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

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

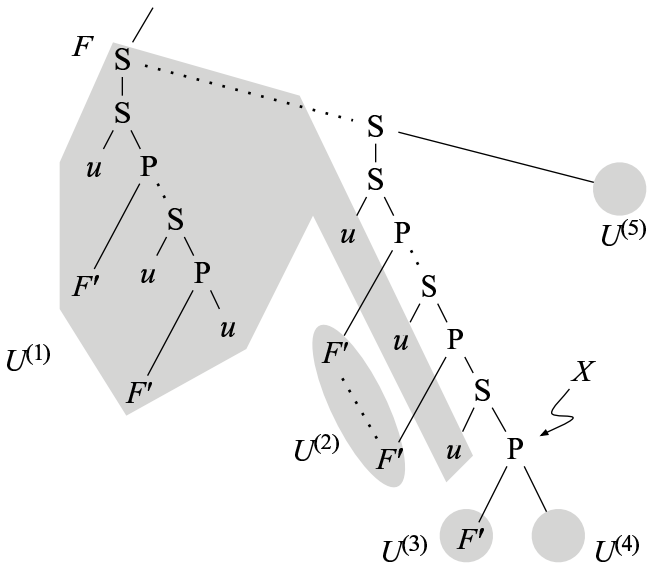

Figure 3-11: The split of a trace U around a P-node X in terms of a canonical Cilk parse tree (see Figure 3-1). The notation F' represents the parse tree of a spawned procedure, and u represents a thread. The tree walk of U is executing in $left[X]$ when the subtree rooted at $right[X]$ is stolen by 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.

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 3-11 is shown in Figure 3-12. 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. Although there is no clear relationship among all the threads in $U^{(1)}$ and $U^{(2)}$, since neither of these traces contains unexecuted threads, SP-hybrid never compares them.

SP-bags can be adapted to implement the local-tier operations LOCAL-INSERT, LOCAL-

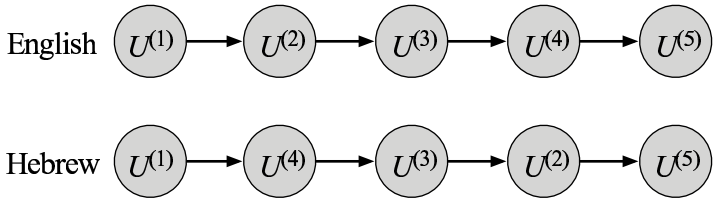

Figure 3-12: An ordering of the new traces resulting from a steal as shown in Figure 3-11. Each circle represents a trace.

PRECEDES, FIND-TRACE, and SPLIT required by SP-hybrid. All 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 SP-bags implementation used by SP-hybrid follows that of Section 3.1, 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 (i.e., the right subtree of the P-node X in Figure 3-11) in the top-level 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. (Although $U^{(4)}$ and $U^{(5)}$ belong to the same procedure as $U^{(1)}$, we modify SP-bags to treat them as new procedures.) Thus, the split can be performed in $O(1)$ time, since only $O(1)$ bookkeeping needs to be done including updating pointers.

We must make one additional change to SP-bags from Section 3.1 to get an efficient SP-hybrid algorithm. The analysis in Section 3.4 bounds the number of steals by arguing that each time a steal occurs, we make headway in the critical path of the computation. Since a UNION is amortized, we may increase the critical path of the computation. In particular, the

```
SP-BAGS(X, F)
     ▷ X is an SP-parse-tree node, and F is a function.
 1   if ISLEAF(X)
 2      then ▷ X is a thread
 3           F ← F ∪ {X}
 4           EXECUTETHREAD(X, F)
 5           return F

 6   if ISSNODE(X)
 7      then F ← SP-BAGS(left[X], F)
 8           return SP-BAGS(right[X], F)

     ▷ X is a P-node
 9   F' ← NEWPROCEDURE()
10   spawn SP-BAGS(left[X], F')
11   if SYNCHED()
12      then P_F ← UNION(F, P_F, S_F')
13      else ▷ A steal occurred.
14           parentstolen[F'] ← TRUE
15           F_return ← NEWPROCEDURE()
16           F ← NEWPROCEDURE()
17   spawn SP-BAGS(right[X], F)
18   sync

19   S_F ← UNION(F, S_F, P_F)
20   P_F ← ∅
21   return F_return
```

Figure 3-13: The local-tier SP-bags algorithm written in parallel pseudocode to operate on the canonical Cilk parsetree from Figure 3-1. This implementation is similar to Figure 3-3 except with the implicit addition of a FAST-MACROUNION where necessary. $parentstolen[F]$ indicates whether the parent procedure of F has been stolen (meaning that F is available to be stolen). Similarly to SP-hybrid, this version of SP-bags returns a function into which future threads should be inserted.

MACROUNION component may take up to $\Theta(n)$ time, where n is the number of microsets. To compensate for this amortization, we also provide a FAST-MACROUNION(x, y), which simply points the representative of the macroset containing y at the representative of the macroset containing x. This operation takes constant time.

We call FAST-MACROUNION(x, y) instead of MACROUNION(x, y) whenever the procedure making the call is ready to be stolen. Specifically, we modify the UNION from Figure 3-6 as follows. We make UNION(F, x, y) take three arguments instead of the original two (x, y). Whenever performing a MACROUNION, the UNION operation performs a constant number of steps of the MACROUNION, then checks whether F is ready to be stolen. If so, we finish with a FAST-MACROUNION. If not, then we perform a constant number of steps and check again. In this way, we guarantee that we never perform more than a constant amount of extra work (due to SP-maintenance) when a thread is waiting to be stolen. Obviously, correctness of the data structure is unaffected by this change, but it is no longer obvious that FINDs are fast. In Section 3.4, we argue that time to perform a FIND is still $O(1)$ in the worst-case.

Figure 3-13 gives parallel pseudocode for the local-tier SP-bags algorithm. This version of SP-bags returns a procedure ID into which future threads should be inserted, which is conceptually similar to SP-hybrid from Figure 3-9. Whenever a steal occurs, as shown in lines 13–16, we create new "procedures" to handle the traces $U^{(4)}$ and $U^{(5)}$. The new procedure, corresponding to trace $U^{(5)}$, is returned to be handled by the appropriate ancestor S-node. This version of SP-bags also uses the FAST-MACROUNION operations where appropriate. Whenever the parent of a procedure F has been stolen, F is available to be stolen, and a FAST-MACROUNION is preferred. We use the field $parentstolen[F]$ to indicate this fact. If we call a UNION(F, x, y), then the UNION periodically checks against $parentstolen[F]$ to see whether it should switch to the FAST-MACROUNION.

Even though Figure 3-13 gives a parallel implementation of SP-bags, the code still reflects a serial algorithm. That is, SP-bags is still only correct if run on a single processor—this version of SP-bags allows parallelization only when run in the context of the local tier of SP-hybrid. Also, as with the SP-bags from Figure 3-3, we do not require any locking. Any particular bag or set is touched by only a single processor, so there is no contention to

worry about.

To attain the full SP-hybrid algorithm, the two parsetree walks from Figures 3-9 and 3-13 must be merged together.

3.3 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 3.2. 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.[8] 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.

Correctness for SP-bags is given in [30]. The only significant difference between our version of SP-bags from Section 3.2 and the version in [30] is that we may spawn new instances of the SP-bags algorithm when a steal occurs. The new instances result from the creation of new bags given in lines 13–16 of Figure 3-13. These instances correspond to the subtraces $U^{(4)}$ and $U^{(5)}$ from Figure 3-11, which are subtrees of the parse tree. Since SP-bags operates correctly on a Cilk-like parsetree, it operates correctly on a subtree as well, and we do not give a new correctness proof here. Instead, we concentrate on correctness of the global tier and the SP-hybrid algorithm as a whole.

The following lemma shows that when a split occurs, the subtraces are consistent with the subtraces properties given in Section 3.2.

Lemma 11 *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,

[8] 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.

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 17–27 of Figure 3-9, 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 $U_i{}^{(3)} = \{u : u \in \mathit{descendants}(\mathit{left}[X])\}$. 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$(\mathit{left}[X], U_i)$ in line 11. If $u \in \mathit{descendants}(\mathit{left}[X])$, 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 $Y = \mathit{parent}(X)$ 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$(\mathit{left}[Y], U_i)$ in line 8 or SP-HYBRID$(\mathit{right}[Y], U_i)$ in line 9. In the first situation, our inductive hypothesis states that SP-HYBRID$(\mathit{left}[Y], U_i)$ 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 14, or else SP-HYBRID$(right[Y], U_i)$ would have already been stolen. We can be only at either line 15 or line 26. If we're at line 15, 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 6–10 and lines 14–16, 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 12 *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 $U_j \in \{U^{(3)}, U^{(4)}, U^{(5)}\}$. 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 11 to conclude that $u_i \prec X$ for some P-node X and $u_j \in \mathit{descendents}(\mathit{left}[X])$, which implies that $u_i \prec u_j$. We also have that $\mathit{Eng}[U^{(1)}] < \mathit{Eng}[U^{(3)}]$ and $\mathit{Heb}[U^{(1)}] < \mathit{Heb}[U^{(3)}]$, which matches the claim.

Case (2,3): $U_i = U^{(2)}$ and $U_j = U^{(3)}$. Lemma 11 allows us to conclude that $u_i \in \{u \in U : u \parallel X \text{ and } u \notin \mathit{descendants}(X)\}$ for some P-node X and that the thread $u_j \in \mathit{descendants}(\mathit{left}[X])$, which means that $u_i \parallel u_j$. We also have that $\mathit{Eng}[U^{(2)}] < \mathit{Eng}[U^{(3)}]$ and $\mathit{Heb}[U^{(2)}] > \mathit{Heb}[U^{(3)}]$, 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 13 *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.

First, consider the case in which no u_i and u_j do not change traces during the execution of SP-PRECEDES. 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 12 shows that the correct result is returned.

We must also show that SP-PRECEDES returns the correct result if the traces for either u_i or u_j are split during the execution of the SP-PRECEDES query. Only a single SPLIT may be in progress at a time because of the global lock used by SP-HYBRID. We consider the state of the traces at the instant in time at which the last SPLIT *completed* before the start of SP-PRECEDES. That is, if there is no SPLIT in progress, then we consider the state of traces at the time SP-PRECEDES begins. If there is a SPLIT in progress when SP-PRECEDES begins, then we consider the state of traces just before *that* SPLIT began.

Suppose that we have u_i and u_j belong to different traces at this time. Consider the code given in Figure 3-10. There is no way to get $U_i = U_j$ in the test in line 30. Moreover, we have that Lemma 12 applies to give us the correct result.

Suppose instead that at the start of the SPLIT, we have u_i and u_j belong to the same trace U. A SPLIT may be in progress. Since u_j is still executing, and it belongs to some trace already, it follows from the subtrace properties that u_j can only be a part of a $U^{(3)} = U$ resulting from a SPLIT. Thus, the trace for u_j cannot change across the execution of SP-PRECEDES. Similarly, u_i can belong one of $\{U^{(1)}, U^{(2)}, U^{(3)}\}$, but it cannot belong to $U^{(4)}$ or $U^{(5)}$, since it already exists at the time of the SPLIT. Given these facts, it does not matter whether we get $U_i = U_j$ in line 30. If $U_i = U_j$ and $u_i, u_j \in U^{(3)}$, then LOCAL-PRECEDES returns the correct result. If $U_i = U_j$ and $u_i \in U^{(1)}$, then LOCAL-PRECEDES still returns the correct result, since u_i is in an S-Bag. Similarly, if $U_i = U_j$ and $u_i \in U^{(2)}$, then LOCAL-PRECEDES returns the correct result, since u_i is in a P-Bag. From Lemma 12, the OM-PRECEDES returns the correct result in either of these two cases. □

3.4 Performance analysis

This section analyzes the SP-hybrid algorithm run on a fork-join program. Suppose that the program has T_1 work and a critical-path length of T_∞. When executed on P processors using a round-robin work-stealing scheduler (that satisfies Property 10), SP-hybrid runs in $O(T_1/P + PT_\infty)$ time in the worst case.

First, we show that the local-tier operations LOCAL-PRECEDES (implemented as a

FIND in the disjoint-sets data structure) takes $O(1)$ time in the worst case. Recall that we modify SP-bags and the disjoint-sets data structure for SP-hybrid in Section 3.2 by introducing this notion of a FAST-MACROUNION. Whereas a macroset from Section 3.1 has all elements pointing at the representative, a macroset resulting from a FAST-MACROUNION does not. As a result, it is not immediately obvious that a FIND takes $O(1)$ time in the worst case. To prove this fact, we exploit the structure of the unions performed by the local tier of SP-hybrid. If a macroset has all elements pointing at the representative, then we say that the depth of the macroset is 1. When the FAST-MACROUNION is performed, the depth may increase, but we bound this increase.

Lemma 14 *For a procedure F that is ready to be stolen in an execution of SP-hybrid, the depth of the macrosets for S_F and P_F are at most 2 and 3, respectively.*

Proof. Proof by induction on the unions involving these bags.

Any child procedure F' is not ready to be stolen, because of Property 10-1 requires parents to be stolen before their children. Thus, since no FAST-MACROUNIONs are performed on F', the depth of $S_{F'}$ is 1 when F' returns to F.

The UNION$(F, P_F, S_{F'})$ is only performed in line 12 of Figure 3-13. Since the modified UNION points the representative of $S_{F'}$ at the representative for P_F, this union does not increase the depth of P_F past 2.

The only point at which we union with S_F is line 19. Since the representative for P_F is pointed to the representative of S_F, and the depth of P_F is at most 2 by the inductive assumption, we have that the depth of S_F does not increase past 3. □

Given Lemma 14, we have that FINDs take $O(1)$ worst-case time, which gives us the following corollary.

Corollary 15 *A* LOCAL-PRECEDES *performed SP-hybrid takes $O(1)$ worst-case time.* □

To prove the desired bound on the entire SP-hybrid execution, we must first bound the number of steals performed. The following theorem, similar to Theorem 9 from [10], applies to our environment when using the Cilk scheduler. We assume that processors are moving at roughly the same speed.

Theorem 16 *Consider any fork-join program with T_1 work and critical-path length T_∞. When executed on P processors using the Cilk scheduler, SP-hybrid has $O(PT_\infty)$ steal attempts in expectation. Moreover, for any $\varepsilon > 0$, the number of steal attempts is $O(P(T_\infty + \lg(1/\varepsilon)))$ with probability at least $1 - \varepsilon$.*

Proof. The proof is virtually identical to one given by Arora, Blumofe, and Plaxton [10]. A sketch of the proof is given here. For the full version, refer to the paper. Arora et al. assign potentials to each ***ready*** thread—those threads that are ready to be executed. In particular, let $d(u)$ be the depth of a thread u, or the distance of u from the start of the computation. Then

$$\phi(u) = \begin{cases} 3^{2(T_\infty - d(u))-1} & \text{if } u \text{ is assigned to some processor;} \\ 3^{2(T_\infty - d(u))} & \text{otherwise.} \end{cases}$$

The potential only decreases over time. Whenever a thread is stolen, it is assigned to a processor. Whenever a processor completes an assigned thread, it enables up to two children which are deeper in the computation. Either of these actions decreases the total potential.

To bound the number of steals, they group potentials by processors owning the threads. The crux of the argument is that whenever a thief processor tries to steal from a victim processor q, the victim loses a constant factor of its potential. It turns out that the next thread stolen from a particular processor q contributes a constant fraction of that processor's potential. Thus, if a successful steal occurs, the potential decreases by a constant fraction of q's potential. Similarly, if q does not have any threads ready to be stolen, then completing the current thread also decreases q's potential by a constant fraction. Therefore, even if the steal attempt is not successful, we know that the potential decreases.

They apply a balls-in-bins argument to argue that each contiguous ***round*** of P steals reduces the total potential by a constant factor with constant probability. A Chernoff bound across the rounds gives a high-probability result.

In our case, the work completed during any step may not be real work towards the original fork-join program. To compensate, we blow up each instruction by a factor of r, where

r is the worst-case cost of the $O(1)$ SP-PRECEDES (without retrying the OM-PRECEDES) queries performed at each instruction and the cost of the local-tier SP-maintenance operation when the thread is ready to be stolen. Note that the above balls-in-bins argument relies on the fact that we make progress towards the critical path when a thread is being worked on or stolen. Thus, we care only about the blowup from SP-maintenance that occurs at this time.

The two SP-maintenance operations that pose the greatest challenge are the UNION, which may need to make a lot of updates, and the OM-PRECEDES operations, which may retry several times. For the former, we note that since FAST-MACROUNION is performed when the thread is ready to be stolen, the blowup (at steal-attempt time) from this operation is at most $O(1)$. As for OM-PRECEDES, recall that we assume processors are moving at the same speed. We assume that the updates performed on a steal take a long enough (constant) amount of time so that an OM-PRECEDES only needs to abort once (if not, we can make SP-hybrid wait for a constant amount of time while holding the global lock, without impacting the asymptotic performance). We, therefore, effectively have a new computation with critical-path length $T'_\infty \leq rT_\infty$, with $r = O(1)$, and the potential and balls-in-bins arguments from Arora et al. still apply. □

Next, we show that the entire SP-hybrid algorithm performs well.

Theorem 17 *Suppose that a fork-join program has T_1 work and a critical-path length of T_∞. When executed on P processors using the Cilk scheduler, SP-hybrid runs in $O(T_1/P + PT_\infty)$ expected time. Moreover, for any $\varepsilon > 0$, SP-hybrid runs in $O(T_1/P + P(T_\infty + \lg(1/\varepsilon)))$ time with probability at least $1 - \varepsilon$.*

Proof. We use an accounting argument similar to [15], 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 have that the expected value of s is $O(PT_\infty)$ from Theorem 16. The seven buckets are as follows:

$\boldsymbol{B_1}$: The work of the original computation excluding costs added by SP-hybrid. We have that $|B_1| = T_1$, 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, $|B_2| = O(s)$.

$\boldsymbol{B_3}$: The work for the local-tier SP-maintenance operations. Since there are $O(1)$ SP-bags operations for each instruction in the computation, and each SP-bags operation costs $O(1)$ amortized time, we have $|B_3| = O(T_1)$.

$\boldsymbol{B_4}$: The waiting time 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 $|B_5| = O(Ps)$.

$\boldsymbol{B_6}$: Steal attempts while the global lock is not held by any processors. We use Theorem 16 to conclude that $|B_6| = O(PT_\infty)$ in expectation, or $|B_6| = O(P(T_\infty + \lg(1/\varepsilon)))$ with probability at least $1 - \varepsilon$.

$\boldsymbol{B_7}$: 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 $O(T_1 + P\,|B_6|)$ at the end of the computation, and hence, dividing by P, we obtain an expected running time of $O(T_1/P + PT_\infty)$ and the corresponding high probability bound. □

It turns out that we can modify the Cilk scheduler to improve the worst-case performance of SP-hybrid. In particular, we modify the scheduler to perform steal attempts in a round-robin order instead of randomly. To perform round-robin steal attempts, we lock a

global list of processors on each steal attempt. Since we lock on successful steals for SP-hybrid anyway, this additional locking does not hurt us. The randomized stealing makes sense in the context of an arbitrary Cilk program, because ordinarily steals are not serialized.

Given the round-robin stealing policy, we can state a worst case bound on the number of steals, similar to Theorem 16.

Theorem 18 *Consider any fork-join program with T_1 work and critical-path length T_∞. When executed on P processors using the round-robin, work-stealing scheduler that obeys Property 10, SP-hybrid has $O(PT_\infty)$ steal attempts in the worst case.*

Proof. We use the same potential function from Theorem 16. We group P contiguous steal attempts into rounds. In a round, a steal attempt occurs on each processor. When attempting to steal from a particular processor, its potential decreases by a constant factor. Thus, in a round, the potential of the entire system decreases by a constant factor. There can be $O(T_\infty)$ such round, for a total of $O(PT_\infty)$ steal attempts in the worst case. □

Applying Theorem 18 to bound the number s of successful steals in Theorem 17, we achieve the following worst-case bound.

Theorem 19 *Consider any fork-join program with T_1 work and critical-path length T_∞. When executed on P processors using the round-robin, work-stealing scheduler that obeys Property 10, SP-hybrid runs in $O(T_1/P + PT_\infty)$ time in the worst case.* □

Corollary 20 *SP-hybrid achieves linear speed-up when $P = O(\sqrt{T_1/T_\infty})$.*

Proof. When $P = O(\sqrt{T_1/T_\infty})$, Theorems 17 and 19 state that SP-hybrid runs in $O\left(T_1/P + \sqrt{T_1/T_\infty} \cdot T_\infty\right) = O\left(T_1/P + T_1/\sqrt{T_1/T_\infty}\right) = O(T_1/P)$. □

Chapter 4

Race Detection

This chapter describes aspects of race detection beyond SP maintenance. Throughout most of this chapter, I assume that the program being tested for races contains no locks. The techniques described in this thesis can be extended to programs that use locks. Section 4.1 describes the access histories necessary for serial and parallel, on-the-fly race detectors. Section 4.2 describes performance of a parallel race-detector that uses the access history from Section 4.1 and SP-hybrid from Section 3.2. Section 4.3 describes how to perform garbage collection to get efficient space usage without asymptotically harming the running time. In particular, I show that the parallel race detector uses $O(Pv)$ space, where P is the number of processors and v is the number of memory locations being monitored.

4.1 Access histories

As introduced in Chapter 1, on-the-fly race detectors maintain two data structures—an SP-maintenance data structure, described in Chapters 2 and 3, and an "access history." This section describes the access history. The approaches described in this section are similar to [17, 30] and [43], for the serial and parallel race detectors, respectively. I begin with a description of an access history used by our serially executing race detector. I then incrementally describe how to extend the access history for a parallel race detector. I describe a parallel access history that locks on every access. Finally, I optimize the parallel access history for SP-hybrid, taking advantage of the structure of traces.

An ***access history*** stores a set of threads that have accessed a given shared-memory location. In a race detector, whenever a particular location is accessed during the program execution, we check the threads in the access history to see if any of those threads operates logically in parallel with the current thread. Since we need to check the currently executing thread against all threads in the access history (for the location), we need to keep the access history small. In particular, to get an efficient race detector, we keep an access history that has $O(1)$ threads for each memory location.

For our serial race detectors, we execute the parse tree in a depth-first (left-to-right) fashion with an access history similar to the one used by Feng and Leiserson's Nondeterminator [30] and Cheng, Feng, Leiserson, Randall, and Stark's [17] Nondeterminator-2. For each memory location l, $reader[l]$ has a thread that previously read l, and $writer[l]$ has a thread that previously wrote l. In particular, we store the "deepest," "leftmost" thread in the dag that has accessed the location. We define a thread u_1 to be ***leftof*** a thread u_2 if $\mathrm{lca}(u_1, u_2)$ is an P-node and u_1 is in the left subtree. This "leftof" relation is consistent with Mellor-Crummey's [43] terminology and the parallel access history introduced later in this section. We say that a thread u_1 is ***deeper*** than a thread u_2 if $u_2 \prec u_1$. A thread u_1 replaces a thread u_2 in the access history if u_1 is deeper than u_2 (i.e., $u_2 \prec u_1$) or if u_1 is leftof u_2 (i.e., $u_1 \parallel u_2$, and u_1 is in the left subtree of the P-node $\mathrm{lca}(u_1, u_2)$). This access-history policy, using a leftof-or-deeper relation, is the same as the one used by Cheng et al.

The leftof-or-deeper relation imposes a partial order on the threads in the parse tree. This partial order is identical to the Hebrew ordering given in Chapter 2. That is to say, a thread u_1 is leftof or deeper than (preceded by) a thread u_2 if and only if u_1 appears later in the Hebrew ordering. Transitivity of the leftof-or-deeper relation naturally follows. The ***deepest, leftmost*** thread is the one that has no other thread deeper than or leftof it. The definition of deepest, leftmost corresponds to taking the thread that appears latest in the Hebrew ordering.

Figure 4-1 gives pseudocode for memory accesses in our serial race detector using SP-order. Suppose that $reader[l] = u'$. Then whenever a thread u reads the location l, we check whether u is leftof or deeper than u', or, equivalently, whether u' precedes u in the Hebrew ordering. Since we execute serially according to an English order, this test in lines 3 and 7

read location l in thread u

```
1  if writer[l] ∥ u
2     then a race occurs
3  if OM-PRECEDES(Heb, reader[l], u)
4     then reader[l] ← u
```

write location l in thread u

```
5  if writer[l] ∥ u or reader[l] ∥ u
6     then a race occurs
7  if OM-PRECEDES(Heb, writer[l], u)
8     then writer[l] ← u
```

Figure 4-1: The serial access-history updates written in serial pseudocode. Each memory location l has a $reader[l]$ and $writer[l]$ that store the latest threads in the Hebrew ordering of the parse tree (the deepest, leftmost threads) that have read and written, respectively, the location l. The Heb data structure gives the Hebrew ordering maintained by SP-order. The $\parallel$ relationships is determined by a call to SP-PRECEDES.

is equivalent to testing $u' \prec u$.[1] If so, then we update the $reader$. We use a similar update policy for $writer$. To check for races, when a write occurs, we check both the $reader$ and $writer$. If either of these threads is logically parallel with the current thread, then a race occurs. On a read, we check just the $writer$ to determine whether a race occurs.

Cheng et al. [17] give correctness of a serial race detector using a similar access-history algorithm.

Parallel access history

The serial, access history from Figure 4-1 does not work correctly for a race detector that runs in parallel. Consider the parse-tree given by Figure 4-2 (a). If the instructions are interleaved as shown in the figure, then at the time of the **write**, we have $reader[l] = u_1$, and no race is detected. Moreover, there is no access-history algorithm that is both correct and stores a single thread in $reader[l]$. Consider both parse trees (a) and (b). If the instructions are interleaved as indicated, then at the time of the **write**, either $reader[l] = u_1$ or $reader[l] = u_2$. In the former case, (a) does not detect the race. In the latter, (b) does not

[1] This same access-history update can be used for race detectors that do not maintain English and Hebrew orderings, as in SP-bags of [30], because $\prec$ is defined by the SP-maintenance algorithm.

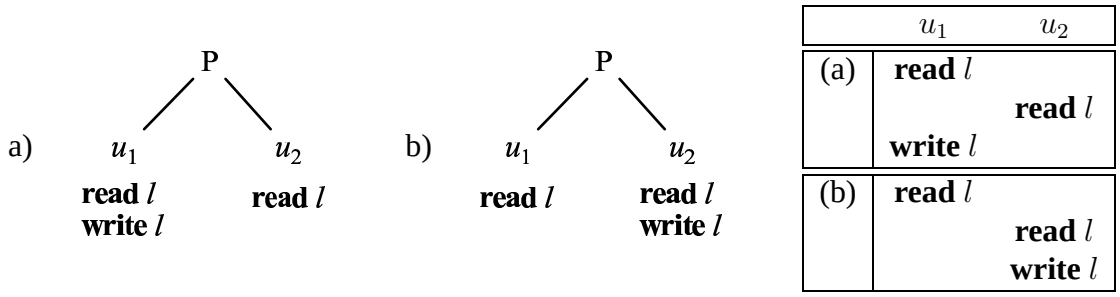

Figure 4-2: Parse trees illustrating the need for a more complex access history for a parallel execution. Under each thread u_i, given by leaves in the parse tree, is the memory access performed by the thread. (a) A parse tree for which the serial access-history algorithm fails on a parallel execution. On the right we have an instruction interleaving that exhibits the problem. (b) Another parse tree. The interleavings shown on the right show that no correct parallel access history has a single $reader$.

detect the race.

For a parallel execution (or a serial execution that does not follow an English ordering), we need an access history that stores two threads that read and wrote the location. In the serial case it is sufficient to store the deepest, leftmost thread (the one occurring latest in the Hebrew ordering). In the parallel case, we also need to store the deepest, ***rightmost*** thread, which is the thread accessing the location that occurs latest in the English ordering. Whereas our serial access history maintains $reader[l]$ that stores the leftmost reader of a location l, the parallel access history maintains $left\text{-}reader[l]$ and $right\text{-}reader[l]$ that store the deepest leftmost and rightmost, respectively, threads reading the location l. Mellor-Crummey [43] introduced this deepest leftmost and rightmost access history. He also proves that maintaining both of these threads is sufficient to detect at least one race on each location involved in a race.

Because there are concurrent threads updating the same access history, we need some sort of mutual exclusion. For now, assume that each access is protected by a lock. This approach is obviously inefficient as we could end up serializing the entire program. I discuss how to better deal with concurrent accesses to the access history at the end of this section.

Figure 4-3 gives pseudocode for memory accesses in our parallel race detector. I show writes only, but reads are similar. Suppose that $left\text{-}writer[l] = u_L$ and $right\text{-}writer[l] = u_R$. Then whenever a thread u writes the location l, we check whether u_L precedes u in the Hebrew ordering (i.e., LEFTOF-OR-DEEPER(u, u_L) returns TRUE) and whether u_R

write location l in thread u

```
1  if left-writer[l] ∥ u or right-writer[l] ∥ u
           or left-reader[l] ∥ u or right-reader[l] ∥ u
2     then a race occurs
3  if LeftOf-or-Deeper(u, left-writer[l])
4     then left-writer[l] ← u
5  if RightOf-or-Deeper(u, right-writer[l])
6     then right-writer[l] ← u
```

Figure 4-3: The parallel (writer) access-history updates written in serial pseudocode. Each memory location l has a $left\text{-}reader[l]$, $right\text{-}reader[l]$, $left\text{-}writer[l]$, and $right\text{-}writer[l]$ that store the latest threads in the Hebrew and English orderings of the parse tree that have read and written the location l. The LEFTOF-OR-DEEPER(u, u') and RIGHTOF-OR-DEEPER(u, u') procedures, given for SP-hybrid in Figure 4-4, determine whether u is leftof or rightof, respectively, or deeper than the thread u'. The $\|$ relationships is determined by a call to SP-PRECEDES.

LEFTOF-OR-DEEPER(u, u')

```
1  U ← FindTrace(u)
2  U' ← FindTrace(u')
3  if U ≠ U'
4     then return OM-Precedes(Heb, U', U)
5     else return Local-Precedes(u', u)
```

RIGHTOF-OR-DEEPER(u, u')

```
6  U ← FindTrace(u)
7  U' ← FindTrace(u')
8  if U ≠ U'
9     then return OM-Precedes(Eng, U', U)
10    else return TRUE
```

Figure 4-4: LEFTOF-OR-DEEPER and RIGHTOF-OR-DEEPER for SP-hybrid, written in serial pseudocode. These operations accept two threads u_1 and u_2, where u is a currently executing thread. LEFTOF-OR-DEEPER and RIGHTOF-OR-DEEPER return TRUE if u' precedes u in the Hebrew and English, respectively, ordering of threads in the parse tree. 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. Eng and Heb maintain the English and Hebrew orderings of traces.

precedes u in the English ordering (i.e., RIGHTOF-OR-DEEPER(u, u_R) returns TRUE).[2] If so, then we update the appropriate writer value. We use a similar update policy for the readers. To check for races, when a write occurs, we check all the readers and writers. On a read, we check just the writers.

Figure 4-4 gives pseudocode for LEFTOF-OR-DEEPER and RIGHTOF-OR-DEEPER operations for the SP-hybrid. Since SP-hybrid maintains the English and Hebrew ordering only between traces, where one of the traces is currently executing, these operations have similar limitations. The LEFTOF-OR-DEEPER(u, u') procedure returns TRUE if u is leftof or deeper than u', where u must be a currently executing thread. If two threads belong to different traces $u \in U$ and $u' \in U'$, with $U \neq U'$, then LEFTOF-OR-DEEPER compares the traces in the Hebrew ordering. If U appears after U' in the ordering, then U is leftof or deeper than U'. If the traces are the same, then we need to use the SP-bags comparison. Recall that within traces, since threads execute serially in the English order, it suffices to compare $u' \prec u$. The RIGHTOF-OR-DEEPER operation is similar, except for the comparisons done within traces. Since the trace executes according to the English ordering, the current thread in the trace is always the deepest, rightmost, executed thread in the trace.

The following lemma states that LEFTOF-OR-DEEPER and RIGHTOF-OR-DEEPER are correct, even across concurrent operations. Given these LEFTOF-OR-DEEPER and RIGHTOF-OR-DEEPER operations, Mellor-Crummey [43] gives correctness of the access history and resulting race detector.

Lemma 21 *Consider any point during the execution of SP-Hybrid on an SP-parse tree. Let u' be a thread that has been discovered, and let u be a thread that is currently executing. Then* LEFTOF-OR-DEEPER(u, u') *returns* TRUE *if and only if u' precedes u in the Hebrew ordering of* all *the threads in the parse tree. Similarly,* RIGHTOF-OR-DEEPER(u, u') *returns* TRUE *if and only if u' precedes u in the English ordering threads.*

Proof. This proof is similar to the proof for Theorem 13. □

We now return to the issue of concurrent updates to the access history. The obvious

[2]The LEFTOF-OR-DEEPER and RIGHTOF-OR-DEEPER operations are not inverses of each other. For example, when a thread u is deeper than a thread u' (i.e., $u' \prec u$), both LEFTOF-OR-DEEPER(u, u') and RIGHTOF-OR-DEEPER(u, u') return TRUE.

```
write location l in thread u
1  UpdateLeftWriter(l, u)
2  UpdateRightWriter(l, u)
3  if left-writer[l] ∥ u or right-writer[l] ∥ u
           or left-reader[l] ∥ u or right-reader[l] ∥ u
4     then a race occurs

UpdateLeftWriter(l, u)
5  if LeftOf-or-Deeper(u, left-writer[l])
6     then Acquire(left-writer-lock[l])
7          if LeftOf-or-Deeper(u, left-writer[l])
8            then left-writer[l] ← u
9          Release(left-writer-lock[l])
```

Figure 4-5: The parallel, writer access-history updates with explicit locking, written in serial pseudocode. This operations are similar to Figure 4-3, except that we do not assume the entire access is protected by a lock. Only the code for a **write** is given. A **read** operation is handled in a similar fashion except that the condition for a race is simpler. The UpdateLeftWriter(l, u) checks updates the $left\text{-}writer[l] \leftarrow u$ if u is leftof the current value stored. This update is protected by a unique lock $left\text{-}writer\text{-}lock[l]$ for each memory location l. The UpdateRightWriter(l, u) procedure (not shown) performs the same operation on the $right\text{-}writer$.

approach is to lock the access history for l each time a read or write of the location l occurs. This approach, however, can be very inefficient. For a program with work T_1 that accesses a single location $\Omega(T_1)$ times, locking results in a running time that is $\Omega(T_1)$.

In many cases, an access to a memory location does not require an update to the access history. In particular, if the current thread u reads the location l, and u is not leftof or deeper than the currently known leftmost, deepest reader $left\text{-}reader[l]$, then there is no reason to update the $left\text{-}reader[l]$. Since the leftof-or-deeper relationship is transitive, no matter how many concurrent accesses happen, the thread u will never be leftof or deeper than the value in the access history, and similarly for the other fields. Locking the access history on these sorts of accesses is unnecessary.

Figure 4-5 gives a slightly more complex version of our access-history update for **write**s with explicit locking. First, we update the access history as necessary with a call to UpdateLeftWriter and UpdateRightWriter. In UpdateLeftWriter (also given in Figure 4-5), we first check whether the current thread u is leftof or deeper than the current

value of $\mathit{left\text{-}writer}[l]$. If not, then we do not update the access history. Otherwise, we acquire a lock on the access history for l and perform the update $\mathit{left\text{-}writer}[l] \leftarrow u$. Because it is possible for another operation to intervene between the first leftof-or-deeper-than comparison in line 5 and the lock acquire in line 6, we check that u is still leftof or deeper than $\mathit{left\text{-}writer}[l]$ in line 7 before performing the final update in line 8. We then perform a similar update for the $\mathit{right\text{-}writer}$. Finally, we check if a race occurs. The updates for **read**s (not shown) are similar except that a race occurs only if a writer is a logically parallel thread.

We next prove that this parallel access-history algorithm is correct, even under concurrent updates to the access history.

Theorem 22 *Consider a fork-join program run in conjunction with the parallel access-history algorithm as given in Figure 4-5. Suppose also that SP-maintenance algorithm correctly supports* LEFTOF-OR-DEEPER, RIGHTOF-OR-DEEPER, *and* $\parallel$. *Then, the access-history algorithm reports a race on the location l if and only if a race exists.*

Proof. ($\Rightarrow$) Suppose that a race is reported. Then, we have that $\mathit{left\text{-}writer}[l]$ and $\mathit{right\text{-}writer}[l]$ are threads that wrote the location l. Similarly, we have $\mathit{left\text{-}reader}[l]$ and $\mathit{right\text{-}reader}[l]$ are threads that read the location l. Thus, if a race is reported, there must be two parallel threads accessing the location, one of which performs a write.

($\Leftarrow$) Suppose that a race exists on the location l. Let u_L be the deepest, leftmost thread involved in a race on location l. Since u_L is involved in a race, there is some thread $u \neq u_L$ with $u \parallel u_L$ such that u and u_L are involved in a race. Since $u \parallel u_L$, they must have different relationships in the English and Hebrew ordering of threads (from Corollary 2). Thus, u is rightof u_L. Let $u_R \neq u_L$ be the deepest, rightmost thread involved in a race with u_L on location l. For clarity, consider the case that u_L writes to l, and u_R reads from l. All the other cases are similar. Consider the point in line 3 when u_L checks for a race. If $\mathit{right\text{-}reader}[l] = u_R$ at the time, then a race is detected, and we are done. Otherwise, if $\mathit{right\text{-}reader}[l] \neq u_R$, then u_R has not yet tested for a race. Thus, when u_R tests for the race, we have $\mathit{left\text{-}writer}[l] = u_L$, and the race is discovered. □

Access history for SP-hybrid

I now give an improvement for the access history to exploit structure given by SP-hybrid. The access history introduces some performance overhead due to the waiting time for the lock acquire in line 6 of Figure 4-5. As a result, we'd like to bound the amount of time spent performing an update (see Corollary 24). Recall that SP-hybrid divides the computation into traces. We modify our access history to take advantage of these traces, enabling access-history updaters to abort more quickly.

One behavior we are trying to avoid is illustrated by the following example. Consider an execution of SP-hybrid that includes P parallel threads $u_1, u_2, \ldots, u_P$ that all try to update $\mathit{left\text{-}writer}[l]$ at the same time. Without loss of generality, we have u_i is leftof u_{i+1} for all $i = 1, 2, \ldots, P-1$. Suppose also that no previous thread has written to l. The scheduling might be such that all of these threads attempt the ACQUIRE on $\mathit{left\text{-}writer\text{-}lock}[l]$ at roughly the same time, but they succeed in the order $u_P, u_{P-1}, \ldots, u_1$. Thus, some of the ACQUIREs introduce $\Theta(P)$ waiting time in this case.

This example both illustrates the problem and motivates our solution. If all P threads arrive at the same time, the only one that really needs to update $\mathit{left\text{-}writer}[l]$ is the leftmost thread u_1. If the rest of the threads can simply discover that some thread to their left is trying to perform an update, they can drop out. Moreover, we want multiple locks so that contention at any single lock is $o(P)$ in the worst case.

To improve the access history, we exploit more of the structure of SP-hybrid. During an execution of the SP-hybrid with the fork-join program, there are (at most) P processors working, and hence P traces are active at any time. Moreover, since all of these traces are logically parallel, the leftof and rightof relationships give a total order. We modify SP-hybrid to keep a balanced, binary search tree (e.g., a red-black tree) maintaining the order of these active traces, with all the traces at the leaves. I call this tree the ***access-history tree***. Whenever a processor successfully steals, SP-hybrid removes its old trace from the tree, performs the normal SP-hybrid work, and inserts its new traces into the tree. Since the "leftof" and "rightof" comparisons are performed by constant-time OM-PRECEDES operations, insertions and deletions to this tree take $O(\lg P)$ worst-case time.

I now describe the final access history, for example, $\mathit{left\text{-}writer}$, given this access-history tree. First, let us assume that the tree is fixed. Later, I will describe how to deal with the tree changing because of steals. Each leaf node U (corresponding to a trace) or internal node X in the fixed tree contains a bit $X.\,\mathit{wrote}[l]$ that indicates whether any node in the subtree rooted at U has written to l. Each node also has a unique lock $X.\,\mathit{left\text{-}writer\text{-}lock}[l]$ for each memory location l. Our update still acquires locks, but the advantage to spreading locks across the tree is that there are never many processors waiting on a lock.

Figure 4-6 gives the UPDATELEFTWRITER that would be performed instead of lines 5–9 of Figure 4-5. In Section 4.2, we show that any update completes in $O(\lg P)$ time in the worst case. When a thread u in trace U writes to l, we do the following. First, we try to update the $\mathit{left\text{-}writer}[l]$ if u and the $\mathit{left\text{-}writer}$ are currently in the same trace. This technique, given in lines 2–7 provides a shortcut for a "common" case. If the thread u performing the write belongs to a different trace from $\mathit{left\text{-}writer}[l]$, we walk up the access-history tree by performing a call to UPDATELEFTWRITERTREE. In particular, we walk up from the node U, acquiring locks $X.\,\mathit{left\text{-}writer\text{-}lock}[l]$ and updating the appropriate values of $X.\,\mathit{wrote}[l]$, for any nodes X encountered along the path to the root. To guarantee that a call completes quickly, we perform a TEST-ACQUIRE in line 12 to acquire these locks. This procedure not only acquires the lock as normal, but it also observes whether it blocks due to a concurrent lock acquisition. In particular, TEST-ACQUIRE returns TRUE if and only if the lock is available at the time of the call, and no concurrent operation acquires the lock first.[3] If at any point a TEST-ACQUIRE fails, then we give up on the tree walk, releasing locks down the tree. If the TEST-ACQUIRE succeeds, we update the node's $\mathit{wrote}[l]$ field in line 14. Before continuing up the tree from a node X, line 20 checks whether X is in the right subtree of its parent and a trace in the left subtree has written to l. If so, then some thread leftof the current thread u is trying to update the access history, and u gives up on the update to avoid interfering. When reaching the root of the tree, given by lines 15–19,

[3]The TEST-ACQUIRE can be implemented using a regular ACQUIRE primitive as follows. Keep a counter associated with each lock. Whenever performing a TEST-RELEASE, first increment the counter then perform the regular RELEASE operation. Whenever performing the TRY-ACQUIRE, first read the value of the counter and check whether the lock is held. If the lock is held, then the return value is FALSE. In any case, next perform a regular ACQUIRE. Once the lock is acquired, check the value of the counter. If the counter has changed (some other acquire and release intervened), then return FALSE. Otherwise, return TRUE.

```
UPDATELEFTWRITER(l, u)
 1  U ← FINDTRACE(u)
 2  if LEFTOF-OR-DEEPER(u, left-writer[l]) and U = FINDTRACE(left-writer[l])
 3     then ACQUIRE(left-writer-lock[l])
 4          if LEFTOF-OR-DEEPER(u, left-writer[l]) and U = FINDTRACE(left-writer[l])
 5            then left-writer[l] ← u
 6          RELEASE(left-writer-lock[l])
 7          return
 8  done ← FALSE
 9  while LEFTOF-OR-DEEPER(u, left-writer[l]) and not done
10        do done ← UPDATELEFTWRITERTREE(treenode[U], l, u)
11  wait until not LEFTOF-OR-DEEPER(u, left-writer[l])

UPDATELEFTWRITERTREE(X, l, u)
12  if TEST-ACQUIRE(X. left-writer-lock[l])
13     then done ← TRUE
14          X. wrote[l] ← TRUE
15          if X = root
16            then ACQUIRE(left-writer-lock[l])
17                 if LEFTOF-OR-DEEPER(u, left-writer[l])
18                   then left-writer[l] ← u
19                 RELEASE(left-writer-lock[l])
20            else if X = left[parent[X]] or parent[X]. wrote[l] = FALSE
                        or left[parent[X]]. wrote[l] = FALSE
21                   then done ← UPDATELEFTWRITERTREE(parent[X], l, u)
22     else done ← FALSE    ▷ a TEST-ACQUIRE failed.
23  TEST-RELEASE(X. left-writer-lock[l])
24  return done
```

Figure 4-6: An access-history update optimized for SP-hybrid, given in serial pseudocode. The UPDATELEFTWRITER procedure is called in line 1 of Figure 4-5. The goal of this procedure is to makes any writer finish an update "quickly" and to give priority to the leftmost writer. The UPDATELEFTWRITERTREE is an auxiliary procedure that acquires locks going up the access-history tree until discovering a writer to the left of the current trace. For a trace U, $treenode[U]$ is the node in the binary search tree. Each node X in the tree has a field $X.\,wrote[l]$ that indicates whether any trace in the subtree rooted at X wrote to l, and a lock $X.\,left\text{-}writer\text{-}lock[l]$ to lock the node for the location l. There is also a unique lock $left\text{-}writer\text{-}lock[l]$ for each memory location l that is acquired before updating $left\text{-}writer[l]$ or $left\text{-}writer\text{-}trace[l]$. The procedures TEST-ACQUIRE and TEST-RELEASE acquire and release a lock, respectively. TEST-ACQUIRE returns TRUE if and only if it does not wait on any other processors while acquiring the lock. In either case, it does acquire the lock. The UPDATELEFTWRITERTREE procedure returns FALSE if any FASTACQUIRE fails.

we acquire the $left\text{-}writer\text{-}lock[l]$ lock and perform the regular update (as in lines 2–7) of $left\text{-}writer[l]$.

When UPDATELEFTWRITERTRACE returns to UPDATELEFTWRITER(l, u), either u tried to update the $left\text{-}writer[l]$ in lines 15–19, the tree walk discovered a thread leftof u that is trying to update the access history, or a TEST-ACQUIRE failed. In the first two cases, UPDATELEFTWRITERTRACE returns TRUE. Otherwise, UPDATELEFTWRITERTRACE returns FALSE. When UPDATELEFTWRITERTRACE fails due to a TEST-ACQUIRE, we retry in lines 9–10. Otherwise, line 11 waits until the $left\text{-}writer[l]$ is u or leftof u.[4]

Correctness still follows from Theorem 22. The algorithm given in Figure 4-6 only makes threads give up on attempting to update $left\text{-}writer[l]$ sooner (so that they don't compete on lock acquisition). The waiting in line 11 linearizes the completion of accesses such that if u doesn't update $left\text{-}writer[l]$, then a thread u' that is leftof u updates $left\text{-}writer[l]$ before u returns from UPDATELEFTWRITER.

Now, I describe how to deal with the fact that a concurrent steal may change the structure of the access-history tree. We could maintain the field $wrote[l]$ across a rotation in a red-black tree, but there are $O(1)$ such fields for each of the memory locations. Since the number of memory locations can be huge, this sort of update would take a long time. In particular, if there are v memory locations, then a single insertion into the tree, while maintaining $wrote[l]$ for all l, may take $\Omega(v \lg P)$ time. Moreover, just initializing a new tree node to have FALSE for every $wrote[l]$ value takes $\Theta(v)$ time.

Instead of maintaining $wrote[l]$ across rotations, we keep a global counter that is incremented whenever the stealing processor starts and stops updating the tree. Rather than setting $wrote[l]$ to TRUE or FALSE in a tree node, we set them to the current value of the counter. If the value $wrote[l]$ is the same as the counter, then we consider the value to be equivalent to TRUE in Figure 4-6. If the counter changes at any point during UPDATELEFTWRITER, then we restart. Thus, as with SP-PRECEDES, we can see whether a concurrent steal invalidates the action. We can also avoid initializing new nodes. Whenever a process inserts a new node into the tree (except the first time), it also removes its old trace.

[4]The thread $left\text{-}writer[l]$ cannot be deeper than u because u hasn't completely executed yet. Thus, we care only about the leftof relation here.

We just reuse the same block of memory, with $wrote[l]$ carrying over from the previous trace—since the counter increments, these values are all invalid and read as FALSE anyway. Using this technique, the maximum value of the counter is bounded by the number of traces created. In Section 4.2, I prove that the number of traces is bounded by $O(PT_\infty)$.

Dealing with locks

The parallel access-history algorithm can be augmented to deal with programs that contain locks. In particular, we are currently working on an implementation of Nondeterminator-3 that incorporates Cheng, Feng, Leiserson, Randall, and Stark's ALL-SETS algorithm [17]. Incorporating this algorithm is straightforward and is therefore not included in this thesis.

The ALL-SETS algorithm maintains an access-history for each location l and set of locks S held while accessing l, e.g., $\textit{left-writer}[l, S]$. Whenever a thread writes the location l holding the locks in the set S, it performs an update, as in Figure 4-6, on $\textit{left-writer}[l, S]$. To check for the existence of a race, the thread u performing the access must check whether the thread $\textit{left-writer}[l, S']$ stored for each lock set S' not sharing any locks with S (i.e., $S \cap S' = \emptyset$) operates logically in parallel with u, and similarly for the other access-history location (e.g., $\textit{right-writer}[l, S']$). Cheng et al. show that in a serial race detector, an access takes $O(n^k)$ time, where n is the number of locks and $k \ll n$ is the maximum number of locks held simultaneously. Parallelizing the ALL-SETS algorithm is straightforward.

If the program follows the "umbrella" locking discipline, whereby all parallel accesses to a variable hold a single lock,[5] Cheng et al. give a more efficient algorithm called BRELLY. Unlike the ALL-SETS algorithm BRELLY keeps only a single lock set for each memory location. In a serial race detector using BRELLY, they show that an access takes $O(k)$ time, where k is the maximum number of locks held at any time. Parallelizing the BRELLY algorithm is more complex and outside the scope of this thesis. Also, since BRELLY reports violations of the umbrella locking discipline, which may not necessarily be data races, BRELLY is not as robust an algorithm as ALL-SETS.

[5]The single lock does not have to be the same across the entire program execution. The umbrella discipline just requires that for every set of threads having a P-node as the least-common ancestor, all threads in the set hold the same lock while accessing the location l.

I do not give the analysis of these augmentations in Section 4.2—that section considers performance only for programs that do not contain locks. These lock set algorithms result in two overheads. First, the cost of a memory access is multiplied by a factor of $O(n^k)$ or $O(k)$ for ALL-SETS and BRELLY, respectively, in the worst case. Second, we may need to require additional mutual exclusion on access-history updates. For example, in the ALL-SETS algorithm, whenever a new lock set is discovered, we need to add it to the list of lock sets for the location. If the possible n locks (and hence n^k lock sets) are known a priori, then updating this list is not an issue, and the performance for ALL-SETS can be obtained straightforwardly by multiply the running time in Theorem 28 by $O(n^k)$ to account for access-history lookup cost.

4.2 Performance analysis

This section analyzes our parallel race detector, called Nondeterminator-3, that uses SP-hybrid from Section 3.2 as the SP-maintenance algorithm and the parallel access history from Section 4.1. We give two bounds different bounds for the same algorithm. The bound that applies depends on the structure of the computation and the number of processors on which the race detector is run. If the number v of memory locations is small, then our race detector runs in $O(T_1/P + (v + P)T_\infty \lg P)$ worst-case time. If the number of memory locations is large, then our race detector runs in $O((T_1/P + PT_\infty) \lg P)$ time.

The serial race detectors, using SP-bags from Section 3.1 or SP-order from Chapter 2, are trivial to analyze. There are no locking overheads, and all the other additional race-detection work is $O(1)$ per instruction in the program. Thus, we have that a serial race detector runs in $O(T_1)$ time, where T_1 is the serial running time of the original program.

For a parallel race detector, the access history introduces additional overheads beyond SP-hybrid, because of waiting time introduced by lock acquisition and retries when updating the binary tree. We first bound the amount of time a single access-history update takes, even when there are multiple concurrent access-history updates. The following lemma and corollary state that any update completes in $O(\lg P)$ worst-case time. To attain this bound, we again assume that processors are moving at the same speed.

Lemma 23 *Consider an* UPDATELEFTWRITER *operation from Figure 4-6. Assuming no interrupting steal, and that processors are moving at the same speed, this operation performs* $O(1)$ *calls to* UPDATELEFTWRITERTREE.

Proof. I show that UPDATELEFTWRITERTREE is called at most twice in lines 9–10 of a single call to UPDATELEFTWRITER.

Consider a call to UPDATELEFTWRITERTREE from the thread u that writes a location l. I claim that by the time the UPDATELEFTWRITERTREE$(treenode[U], l, u)$ returns, we have $X.\,wrote[l] =$ TRUE for every node X between $treenode[U]$ and the root. I prove this claim by induction on the ancestors of $treenode[U]$ and the TEST-ACQUIRES performed at a node. The UPDATELEFTWRITERTREE procedure stops ascending the tree only if some TEST-ACQUIRE at a node X fails. A TEST-ACQUIRE$(X.\,left\text{-}writer\text{-}lock[l])$ can only fail if some other concurrent TEST-ACQUIRE$(X.\,left\text{-}writer\text{-}lock[l])$ obtained the lock first, and the failed TEST-ACQUIRE blocks until the concurrent operation completes. By induction, by the time the concurrent call completes, the path from $parent[X]$ to the root has been updated. As a base case, the first TEST-ACQUIRE$(X.\,left\text{-}writer\text{-}lock[l])$ to obtain the lock succeeds, thereby marking $X.\,wrote[l]$ and continuing up the tree.

Consider the second call to UPDATELEFTWRITERTREE$(treenode[U], l, u)$. Since the first call updates a path from the leaf to the root, any calls concurrent with the second observe in line 20 that U is trying to update the writer, and hence they abort.[6] Thus, the only threads competing with u for a lock are leftof u, and hence by the time this second call completes, either $left\text{-}writer[l] = u$, $left\text{-}writer[l]$ is leftof u, or u has discovered some concurrent writer leftof it in line 20. In any of these cases, the while loop in lines 9–10 terminates. □

Corollary 24 *Consider an* UPDATELEFTWRITER *operation from Figure 4-6. Assuming no interrupting steal, this operation completes in* $O(\lg P)$ *time regardless of the number of*

[6]The exception is concurrent calls that have already completed the appropriate line 20 on u's first tree ascent, but not yet performed the TEST-ACQUIRE on an ancestor of $treenode[U]$. Recall that we assume processors are moving at the same speed. Thus, these calls all block on the TEST-ACQUIREs on ancestors of $treenode[U]$. These TEST-ACQUIREs then acquire and release the lock (in constant time, due to the fact that they blocked) while a tree descent releases locks on the way down the tree. Thus, by the time the second UPDATELEFTWRITERTREE$(treenode[U], l, u)$ begins, these troublesome instances have already passed the point of being a problem.

concurrent operations.

Proof. The two factors contributing to running time are the actual work performed by the thread u performing the update, and the time spent waiting on a lock in a TEST-ACQUIRE or ACQUIRE. By Lemma 23, the actual work is bounded by $O(\lg P)$. We therefore consider time spent waiting on locks.

Since the access history tree is a binary tree, and UPDATELEFTWRITERTREE only acquires locks on a node after acquiring a lock on of its children, there can be at most two processors waiting on any particular lock. Thus, since locks acquisitions queue, the waiting time on a lock at node X is just the time it takes for an UPDATELEFTWRITERTREE to return (and release the lock) from the node X. I claim that a lock at depth d is released in $O(d)$ time, by induction. For a base case, consider a call from the root of the tree in lines 15–19 after TEST-ACQUIRE succeeds. We wait to acquire the global lock $\mathit{left\text{-}writer\text{-}lock}[l]$. The only other thread acquiring this lock concurrently is the one in the same trace as $\mathit{left\text{-}writer}[l]$ in lines 2–7. Thus, this operation completes in constant time. If the TEST-ACQUIRE fails, then we are waiting on an operation that completes in constant time. For any call at depth d, we may ascend the tree (doing $O(1)$ work at each level) until a TEST-ACQUIRE fails at height $d' < d$. By inductive assumption, the TEST-ACQUIRE completes in $O(d')$ time, and hence the full UPDATELEFTWRITERTREE returns in $O(d)$ time.

Since the access-history tree has height $O(\lg P)$, the (at most) $O(1)$ calls to the procedure UPDATELEFTWRITERTREE (from Lemma 23) take $O(\lg P)$ time, including the waiting time. The only other contributor to time is the waiting in line 11. I have already argued that if some thread u gives up on the UPDATELEFTWRITERTREE, then only threads leftof u will complete on the next try. Since some thread makes it to the root in $O(\lg P)$ time, it follows that we wait for at most $O(\lg P)$ time in line 11. □

In UPDATELEFTWRITER of Figure 4-6, we include the shortcut for the common case (lines 2–7) to update the $\mathit{left\text{-}writer}[l]$ if the trace hasn't changed. Since UPDATELEFTWRITERTREE is not called if the shortcut successfully updates $\mathit{left\text{-}writer}[l]$, this update takes $O(1)$ time. The following corollary bounds the total amount of time spent performing all the UPDATELEFTWRITERs and takes advantage of the shortcut cost. This corollary

ignores the cost of restarting an update due to a steal.

Corollary 25 *For an execution of SP-hybrid with the access history from Figure 4-6 that results in n traces, at most $O(n)$ access-history updates for each location l take $\Omega(\lg P)$ time. All other access-history updates take $O(1)$ time.*

Proof. The leftof-or-deeper and rightof-or-deeper relationship between two traces never changes. This fact follows from the fact that traces do not move in the order-maintenance data structures. We charge one slow ($\Omega(\lg P)$) time) update against each trace.

The first time a trace U tries to update the access history, it performs the tree update in UPDATELEFTWRITERTREE. Once it finishes, either $U = \text{FINDTRACE}(\textit{left-writer}[l])$, or $\textit{left-writer}[l]$ is leftof U. In the latter case, since no thread in U will ever be leftof $\textit{left-writer}[l]$, all future calls to UPDATELEFTWRITER on the memory location l from the trace U complete in constant time. In the former case, subsequent updates take $O(1)$ time via lines 2–7, until $\textit{left-writer}[l]$ is no longer in the trace U. If $\textit{left-writer}[l]$ changes because a concurrent thread updates $\textit{left-writer}[l]$, then U will never be leftof $\textit{left-writer}[l]$ again. If $U' = \text{FINDTRACE}(\textit{left-writer}[l])$ changes because of a steal, we charge U's (existing) slow UPDATELEFTWRITER against the subtrace U'. Recall from the subtrace properties of SP-hybrid that $U^{(1)}$ and $U^{(2)}$ are fully expanded (and thus inactive), $U^{(3)} = U$, and $U^{(4)}$ and $U^{(5)}$ are empty, and hence no future UPDATELEFTWRITERs will be charged against this subtrace. Thus, we charge at most 1 slow UPDATELEFTWRITER against each trace for each memory location l. □

I now bound the number of steal attempts when our race detector Nondeterminator-3 is run on a (Cilk-like) scheduler that obeys Property 10, with randomized or round-robin work stealing.

Lemma 26 *Suppose that a fork-join program has T_1 work and a critical-path length of T_∞. When executed on P processors using the Cilk scheduler, the Nondeterminator-3 has $O(PT_\infty \lg P)$ steal attempts in expectation. Moreover, for any $\varepsilon > 0$, the number of steal attempts is $O(P(T_\infty \lg P + \lg(1/\varepsilon)))$.*

Proof. Apply Theorem 16 with one modification. Rather than blowing up each thread by $r = O(1)$ work, we blow up each thread by $r = O(\lg P)$, which is the worst-case

blowup of a memory access. As in Theorem 16, we assume that processors are moving at the same speed, and a steal holds the lock for long enough (an appropriate constant in front of $\Omega(\lg P)$) that an access-history update only aborts one time. Thus, each time a steal attempt happens, we make progress towards the critical path $T'_\infty = O(T_\infty \lg P)$. □

For simplicity, let's modify the round-robin work stealing to wait for $\Theta(\lg P)$ time between each round of P steal attempts. This modification is not strictly necessary to attain the performance bounds given below, but it does simplify the analysis.[7]

Lemma 27 *Suppose that a fork-join program has T_1 work and a critical-path length of T_∞. When executed on P processors using the round-robin, work-stealing scheduler that obeys Property 10, the Nondeterminator-3 has $O(PT_\infty)$ steal attempts in the worst case.*

Proof. We use the same proof as from Theorem 18. Since we wait $\Theta(\lg P)$ time before starting each round of steals, any thread on a processor from which we attempted to steal from in the previous round, even with the $O(\log P)$ blowup from a memory access, has had time to complete. Thus, when a steal attempt occurs, by the end of the round, the potential on that processor has decreased by a constant fraction. Therefore, there can be at most $O(PT_\infty)$ steal attempts, as before. □

I now bound the time taken by the Nondeterminator-3 when run on a Cilk-like scheduler. I have two incomparable bounds that both apply—the one to choose depends on the number of memory locations being monitored. If the number v of memory locations is small, then the Nondeterminator-3 runs in $O(T_1/P + (v+P)T_\infty \lg P)$ time. If the number of memory locations is large, then the Nondeterminator-3 runs in $O((T_1/P + PT_\infty) \lg P)$ expected time.

Theorem 28 *Suppose that a fork-join program has T_1 work, a critical-path length of T_∞, and v shared-memory locations.*

When executed on P processors using the Cilk scheduler, the Nondeterminator-3 runs in $O(T_1/P + PT_\infty \lg^2 P + \min\left\{(T_1 \lg P)/P, vT_\infty \lg^2 P\right\})$ expected time. Moreover, for any

[7] Without this modification, we would use something like Lemma 26 to bound the number of steal attempts (that take $O(1)$ time) and a variation of the Lemma 27 to bound the number of successful steals.

$\varepsilon > 0$, *the Nondeterminator-3 race detector runs in* $O(T_1/P + P(T_\infty \lg P + \lg(1/\varepsilon)) \lg P + \min\{T_1 \lg P/P, v(T_\infty \lg P + \lg(1/\varepsilon)) \lg P\})$ *time with probability at least* $1 - \varepsilon$.

When executed on P *processors using the round-robin, work-stealing scheduler, the Nondeterminator-3 runs in* $O(T_1/P + PT_\infty \lg P + \min\{(T_1 \lg P)/P, vT_\infty \lg P\})$ *worst-case time.*

Proof. We use the same approach as in Theorem 17, except that we add three new buckets. The new buckets are as follows:

$\boldsymbol{B_8}$: The work spent performing access-history updates. A processor places a dollar in the bucket B_8 while performing an access-history update. We do not include money for accesses that restart due to concurrent steals. Note that we include time spent performing the included OM-PRECEDES but not time spent retrying these OM-PRECEDES operations—those retries are still included in bucket B_5. If there are s steals, then Corollary 25 implies that at most $O(vs)$ updates take $O(\lg P)$ time. Otherwise, Corollary 24 states that each of at most T_1 accesses costs $O(\lg P)$ in the worst case. Thus, we have $|B_8| = \min\{vs \lg P, T_1 \lg P\}$.

$\boldsymbol{B_9}$: The work spent retrying access-history updates due to concurrent steals. Since a steal can cause at most $O(1)$ updates to fail on each processor, and the number of steals is s, we conclude that $|B_9| = O(sP \lg P)$.

$\boldsymbol{B_{10}}$: The work spent updating the access history's search tree while the global lock is held. Since we do $O(\lg P)$ work per steal, we have $|B_{10}| = O(s \lg P)$.

Note that B_4, the waiting time for the global lock, also changes. Whereas we had $|B_4| = O(Ps)$ in Theorem 17, we now have $|B_4| = O(sP \lg P)$ due to bucket B_{10}.

Summing over all buckets and dividing by P, we have that the total money is $O(T_1/P + s \lg P + \min\{(vs \lg P)/P, T_1 \lg P/P\}$. The $\min$ arises from bucket B_8. Substituting in s from Lemmas 26 and 27 proves the theorem. □

4.3 Space requirements

This section describes the space usage of our parallel race detector Nondeterminator-3. I first discuss some garbage-collection techniques to remove stale threads and traces from

SP-hybrid. Then, I argue that the race detector, with garbage collection, has efficient space usage. In particular, consider a fork-join program that contains v memory locations and has a maximum procedure-nesting depth (or P-node-nesting depth) of d. Then the Nondeterminator-3 uses $O(P(v+d))$ space when run on P processors. Note that the d arises from the maximum number of outstanding partially completed procedures. Since a program execution requires some space on the stack for these procedure instances anyway, we can think of the asymptotic increase by our race detector as $O(Pv)$.

SP-hybrid, as described in Section 3.2, keeps an object for each thread or procedure. For a race detector, we do not need to keep threads that have completed and are no longer in the access history. The main difficulty is in garbage collecting these objects safely.

We have each processor keep ownership of all the bags (as in SP-bags) that it creates and all the threads that it executes. In particular, we keep a list of completed threads owned by a procedure. Since ownership of threads is clear, we can also have this processor handle reference counting for procedures and bags without any locking. Only the owning processor can free the memory for these completed objects. We use a deamortized mark-and-sweep garbage collector [37, 42]. The mark-and-sweep garbage collector for SP-bags, as described in [30], proceeds as follows for a particular processor:

1. Iterate through all the (completed) threads owned by the processor, and unmark them.
2. Iterate through the v memory locations in the shadow space, marking threads that are in the shadow space.
3. Iterate through all threads freeing those that are still unmarked. Also free any bags or procedures that have no deallocated threads.

There are only $O(v)$ locations in the shadow space. Thus, if we run this garbage collector every v steps, the number of unfreed, completed threads (and procedures and bags) never exceeds $O(v)$. We can deamortize this technique by executing a constant number of steps on each race-detector instruction.

It is not immediately obvious how deallocating procedures impacts the microsets from SP-bags of Section 3.1. One simple technique is keep a count of the number of procedures in a microset. We do not actually deallocate the procedure until the size of the microset

drops by half. At this point, we compress the microset (remove all the holes) to the left of the bitstring and free any procedures that can be freed. We can deamortize this technique in the same way we deamortize the MAKE-SET for incremental set union in Section 3.1. That is, we keep two copies of a microset. Whenever the size of the microset reduces by half, we allocate a new word for the microset. On every subsequent deletion of a procedure from the microset, we examine a constant number of slots in the old microset, deleting or copying procedures to the new microset as necessary. The constant is chosen such that copy completes before the size of the new microset reduces by half again. In this way, we guarantee that if there are k procedures either outstanding or represented by the access history, then the number of unfreed procedures is $O(k + v)$.

Given these deamortized techniques, we perform a constant number of garbage-collection steps for each race-detection step, and performance is not asymptotically affected. Moreover, we guarantee a small number of threads, procedures, and bags being stored for each processor.

It remains to describe how to garbage collect traces, since the traces belong to a shared data structure. The processor that owns the threads in the trace is responsible for reference counting the trace without locking. Before this processor can free the trace, it must acquire the global lock on the shared order-maintenance data structures. These lock acquisitions have no asymptotic affect on performance—we can charge the overheads incurred by locking when deallocating a trace against the creation of the trace.

Since the number of outstanding procedures and threads owned by a particular processor never exceeds the maximum nesting depth d, we have that a processor never owns more than $O(v + d)$ objects. There are P processors, so we have $O(P(v + d))$ objects in total.

Finally, we must deal with the effect of deallocating objects on concurrent queries. A common technique is ***reference counting*** (from [19, 37], for example), whereby a reader increments a reference counter on the object (i.e., a thread in the shadow space), performs the operation (i.e., the SP-PRECEDES query), then it decrements the counter. Since an object gets deallocated only when its reference counter drops to 0, the object is guaranteed to be around for the duration of the query. While this approach does guarantee that no object gets deleted while being read, it requires some sort of mutual exclusion to increment

a reference count in parallel.

Instead of reference counting, we repeat each step of a query several times to guarantee that the objects are still valid. In particular, consider the result of a memory access. We may end up comparing a thread u' stored in the access history against the current thread u with an SP-PRECEDES query. As long as the thread u' is still in the access history, its bag still exists, and its trace is still in the order-maintenance data structures, etc. Thus, when following each pointer, we check that u' is still in the access history. In this way, we double the constant amount of time needed for a SP-PRECEDES. Since it takes $\Omega(v)$ time before a current thread will be collected, and we assume the processors move at the same speed, an SP-PRECEDES only fails once due to a concurrent deallocation of a thread.

Chapter 5

Related work

This chapter summarizes related work on race detection, SP-maintenance algorithms and order-maintenance data structures.

Static race detectors [8, 11, 16, 29, 44, 56] analyze the text of the program to determine whether a race occurs. Static analysis tools may be able to determine whether a memory location can be involved in a race for any input. These tools, however, are inherently conservative (and report races that do not exist), since the static debuggers cannot fully understand the control-flow and synchronization semantics of a program. For example, dynamic control-flow constructs (e.g., a fork statement) inside loops are particularly difficult to deal with. Mellor-Crummey [44] proposes using static tools as a way of pruning the number of memory locations being monitored by a dynamic race detector.

Dynamic race detectors execute the program given a particular input. Some dynamic race detectors perform a post-mortem analysis based on program-execution logs [18, 28, 35, 45–48], analyzing a log of program-execution events after the program has finished running. On-the-fly race race detectors, like the one given in this thesis, report races during the execution of the program. Both dynamic approaches are similar and use some form of SP-maintenance algorithm in conjunction with an access history. On-the-fly race detectors benefit from garbage collection, thereby reducing the total spaced used by the tool. Post-mortem tools, on the other hand, must keep exhaustive logs.

Netzer and Miller [49] provide a common terminology to unify previous work on dynamic race detection. A ***feasible*** data race is a race that can actually occur in an execution of

the program. Netzer and Miller show that locating feasible data races in a general program is NP-hard. Instead, most race detectors, including the one given in this thesis, deal with the problem of discovering ***apparent*** data races, which is an approximation of the races that may actually occur. These race detectors, like the one in this thesis, typically ignore data dependencies that may make some apparent races infeasible, instead considering only explicit coordination or control-flow constructs (like forks and joins). As a result, these race detectors are conservative and report races that may not actually occur.

Dinning and Schonberg's "lock-covers" algorithm [27] detects apparent races in programs that use locks. Cheng, Feng, Leiserson, Randall, and Stark generalize this algorithm with their ALL-SETS algorithm [17]. Cheng et al. also give a more efficient algorithm, called BRELLY, that can be used if the program obeys an "umbrella" locking discipline. These algorithms can be incorporated into access-history algorithms like the one given in this thesis.

Savage, Burrows, Nelson, Sobalvarro, and Anderson give an on-the-fly race detector called Eraser [51] that does not use an SP-maintenance algorithm, and hence reports races between threads that operate in series. Their Eraser tool works on programs that have static threads (i.e., no nested parallelism) and enforces a simple locking discipline. A shared variable must be protected by a particular lock on every access, or they report a race. The BRELLY algorithm [17] is a generalization of Eraser's locking discipline that can be incorporated into race detectors, like the Nondeterminator-3 given in this thesis, that support nested parallelism and maintain SP relationships. By keeping track of SP relationships, we can report fewer spurious races.

Nudler and Rudolph [50] introduced the English-Hebrew labeling scheme for their SP-maintenance algorithm. Each thread is assigned two static 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 [43] 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. Both of these approaches perform local decisions on thread creation

to assign static labels. Although these approaches result in no locking or synchronization overhead for SP-maintenance, the large labels can outweigh the waiting time experienced by our algorithm.

Dinning and Schonberg's "task recycling" algorithm [26] uses a centralized data structure to maintain series-parallel relationships. Each thread (block) is given a unique task identifier, which consists of a task and a version number. A task can be reassigned (recycled) to another thread during the program execution, which reduces the total amount of space used by the algorithm. Each thread is assigned a parent vector that contains the largest version number, for each task, of its ancestor threads. Similar to SP-bags and SP-hybrid, but unlike English-Hebrew or offset-span labelings, the task-recycling algorithm can only determine the SP relationship between two threads if one of the threads is active. To query the relationship between an active thread u_1 and a thread u_2 recorded in the access history, task recycling simply compares the version number of u_2's task against the version numbered stored in the appropriate slot in u_1's parent vector, which is a constant-time operation. The cost of creating a new thread, however, can be proportional to the maximum logical concurrency. Dinning and Schonberg's algorithm also handles other coordination between threads, like barriers, where two parallel threads must reach a particular point before continuing. Whereas the SP-hybrid algorithm given in this thesis is more efficient for strictly fork-join programs, task-recycling is still very promising for more general programs.

The first order-maintenance data structure was published by Dietz two decades ago [21]. It supports insertions and deletions in $O(\lg n)$ amortized time and queries in $O(1)$ time. Tarjan observed [23] that updates could be supported in $O(1)$ amortized time, and the same result was obtained independently by Tsakalidis [58]. Dietz and Sleator [23] 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 [12] gave two simplified data structures whose asymptotic performance matches the data structures from [23]. Their paper also presents an implementation study of the amortized data structure.

A special case of the order-maintenance problem is the ***online list-labeling problem*** [7, 22, 24, 36], also called the ***file maintenance problem*** [59–62]. 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 [22, 24]. 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, 38] and distance queries used for routing [2, 4, 9, 34, 39, 57]. 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.

Chapter 6

Conclusion

In this thesis, I have presented a new serial SP-maintenance algorithm called SP-order, an improvement to the serial SP-bags algorithm, and a new parallel SP-maintenance algorithm called SP-hybrid. I have given performance analysis of these algorithms alone and when incorporated into an efficient race detector.

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, I 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. I also believe that provably efficient algorithms are scientifically interesting in their own right.

As far as I know, this is the only work that has analyzed the total, asymptotic, worst-case running time of an efficient, on-the-fly race detector. Most other works either perform empirical studies or analyze the worst-case cost of a single operation once a lock has been acquired. I am interested in seeing whether the analysis technique used in this thesis can be extended to other parallel applications. In some sense, we amortize the cost of the (protected) accesses to a shared data structure against the critical path of the underlying computation.

This thesis has not concentrated on race detection for programs that contain locks. I

briefly mentioned how these race detectors can be attained, but I have not done a full performance analysis. It is unclear whether our analysis technique can be applied to the ALL-SETS and BRELLY algorithms to show speedup, in the worst case, over the serial race detector.

Some race detectors, like Dinning and Schonberg's task-recyling algorithm [26], work on programs that contain coordinating constructs between threads other than fork and join. Since these programs cannot be represented as series-parallel parse trees, the SP-maintenance algorithms given in this thesis do not apply. Dinning and Schonberg also give a "coordination list" technique to extend Nudler and Rudolph's English-Hebrew labeling [50] to support these same constructs, but the technique seems inefficient. Naturally, this same technique should be applicable to the SP-maintenance algorithms given in this thesis, but it is unclear that one cannot do better. Can SP-hybrid be extended to efficiently support other forms of coordination between threads?

Throughout this thesis, I assume that parallel reads complete in constant time, even if all the processors access the same memory location at the same time. I do not model the underlying congestion in the memory system. It is unclear how to implement a congestion-free parallel read in a real machine in a scalable way. Instead, we may want to augment our model to include the read congestion. If processors are connected in a tree (or any other $O(\lg P)$-diameter constant-degree network), then parallel reads should be implementable in $O(\lg P)$ time. Thus, we can simply take our performance bounds and blow up by an $O(\lg P)$ factor. This bound is somewhat pessimistic, however, because it seems unlikely that a parallel program would always have all the processors reading the same location at the same time. Is there a better way to model read contention?

Finally, I was surprised to see that using a round-robin work-stealing scheduler resulted in a better worst-case running time for our race detector than the randomized work-stealing scheduler. When running a program that does not serialize steals as SP-hybrid does, the randomized work stealer performs better. Is there some balance that can be achieved here? A Cilk program (that does not contain locks) with T_1 work and critical-path length T_∞ runs on the randomized work-stealing scheduler in $O(T_1/P + T_\infty)$ time in expectation. Is there a scheduler that results in a running time like $O(T_1/P + P^\varepsilon T_\infty)$ in the worst case?

Appendix A

Hash function for microsets

This section describes one viable hash function for STRINGTONUM given in Section 3.1. The hash function I give in this section is a perfect hash, whereby the worst-case search cost is $O(1)$ [20, Section 11.5].[1]

Since the set of keys is deterministic (i.e., w^{w-k-1} for valid k), we deterministically construct the hash function. We let $p = 3^i$ be the smallest power of 3 such that $p \geq 3/2 \min\{w, n\}$. Then the function is quite simply $h(j) = j \bmod p$. The range of the hash function is obviously $\Theta(\min\{w, n\})$. The following lemma implies that the given hash function h maps all $\min\{w, n\}$ keys to different values.

Lemma 29 *Let a, b, and c be positive integers. For a given value of c, there is no solution to $2^a - b3^c = 1$ with $a < 2 \cdot 3^{c-1}$.*

Proof. Note that Euler's phi function [20, p.865] implies that there is a solution with $a \leq 2 \cdot 3^{c-1}$.

We prove the lemma by induction. For a base case of $c = 1$, we have $2^a - b3 = 1$. Obviously, the smallest value of a that yields a solution (with a and b integers) is $a = 2$.

[1] There are many other ways to obtain a constant-time STRINGTONUM in the random-access machine [20]. For example one could hardcode particular hash functions that work for particular word sizes. One could also use a two-level perfect-hashing technique [31] (made dynamic by Dietzfelbinger et al. [25]). This approach, however, makes the construction of the hash table expected amortized constant, whereas the function given in this section gives a worst-case cost. Alternatively, Leiserson, Prokop, and Randall [41] give a technique that uses de Bruijn sequences along with multiplication and a bit shift. This technique may be preferred because it does not require a modulo/division operation.

Assume for the sake of contradiction that there is a solution for the equation with $c = k$ and with $a < 2 \cdot 3^{k-1}$. Then we can rewrite a to be of the form $a = d(2 \cdot 3^{k-2}) + x$, where $0 \leq d \leq 2$, and $x < 2 \cdot 3^{m-2}$. Thus, we have that

$$\left(2^{2 \cdot 3^{k-2}}\right)^d 2^x - b3^k = 1$$

has a solution. Since $a = 2 \cdot 3^{k-2}$ yields a solution to $2^a - b_{(1)}3^{k-1} = 1$, we can rewrite this equation as

$$\left(1 + b_{(1)}3^{k-1}\right)^d 2^x - b3^k = 1 \ .$$

This equation, however, implies that there is a value of b that solves $2^x - b3^{k-1} = 1$, with $x < 2 \cdot 3^{k-2}$, which contradicts the inductive hypothesis. □

Corollary 30 *The function* $h(2^a) = 2^a \mod 3^c$ *yields different values for any* $2 \cdot 3^{c-1}$ *consecutive values of* a*.* □

Bibliography

[1] Serge Abiteboul, Haim Kaplan, and Tova Milo. Compact labeling schemes for ancestor queries. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 547–556. Society for Industrial and Applied Mathematics, 2001.

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

[3] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 258–264. ACM Press, 2002.

[4] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Direct routing on trees. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 342–349. Society for Industrial and Applied Mathematics, 1998.

[5] Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 947–953. Society for Industrial and Applied Mathematics, 2002.

[6] Richard J. Anderson and Heather 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] Arne Andersson and Ola Petersson. Approximate indexed lists. *Journal of Algorithms*, 29:256–276, 1998.

[8] W. F. Appelbe and C. E. McDowell. Anomaly reporting: A tool for debugging and developing parallel numerical algorithms. In *Proceedings of the 1st International Conference on Supercomputing Systems*, pages 386–391. IEEE, 1985.

[9] Marta Arias, Lenore J. Cowen, and Kofi A. Laing. Compact roundtrip routing with topology-independent node names. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, pages 43–52. ACM Press, 2003.

[10] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, New York, NY, USA, June 1998. ACM Press.

[11] Vasanth Balasundaram and Ken Kennedy. Compile-time detection of race conditions in a parallel program. In *ICS '86: Proceedings of the 3rd International Conference on Supercomputing*, pages 175–185, New York, NY, USA, 1986. ACM Press.

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

[13] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 133–144, Barcelona, Spain, June27–30 2004.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

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

[16] David Callahan and Jaspal Sublok. Static analysis of low-level synchronization. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, New York, NY, USA, 1988. ACM Press.

[17] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 298–309, Puerto Vallarta, Mexico, June 28–July 2 1998.

[18] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, 1991.

[19] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

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

[22] Paul F. Dietz, Joel I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Algorithm Theory—SWAT '94: 4th Scandinavian Workshop on Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142. Springer-Verlag, 6–8 July 1994.

[23] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, New York City, May 1987.

[24] Paul F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*. Springer, 11–14 July 1990.

[25] Margin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[26] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM Press, 1990.

[27] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM Press, May 1991.

[28] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '91*, pages 580–588, November 1991.

[29] Perry A. Emrath and Davis A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison, Wisconsin, May 1988.

[30] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June22–25 1997.

[31] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[32] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[33] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[34] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 210–219, 2001.

[35] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Analyzing traces with anonymous synchronization. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II70–II77, August 1990.

[36] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In S. Even and O. Kariv, editors, *Proceedings of the 8th Colloquium on Automata, Languages, and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, July13–17 1981.

[37] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wily & Sons, 1996.

[38] Haim Kaplan, Tova Milo, and Ronen Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 954–963. Society for Industrial and Applied Mathematics, 2002.

[39] Michal Katz, Nir A. Katz, Amos Korman, and David Peleg. Labeling schemes for flow and connectivity. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 927–936. Society for Industrial and Applied Mathematics, 2002.

[40] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[41] Charles E. Leiserson, Harald Prokop, and Keith H. Randall. Using de Bruijn sequences to index a 1 in a computer word. Available on the Internet from `http://supertech.csail.mit.edu/papers.html`, 1998.

[42] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[43] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33. IEEE Computer Society Press, 1991.

[44] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, San Diego, California, May 1993. ACM Press.

[45] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 135–144, Atlanta, Georgia, June 1988.

[46] Robert H. B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *Proceedings of the 1992 International Conference on Parallel Processing*, St. Charles, Illinois, August 1992.

[47] Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II: 93–97, August 1990.

[48] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *PPOPP '91: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 133–144, New York, NY, USA, 1991. ACM Press.

[49] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

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

[51] Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 27–37, New York, NY, USA, October 1997. ACM Press.

[52] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*, November 2001. Available on the Internet from `http://supertech.csail.mit.edu/cilk/manual-5.3.2.pdf`.

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

[54] Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the Association for Computing Machinery*, 26(4):690–715, October 1979.

[55] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[56] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):361–376, 1983.

[57] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10. ACM Press, 2001.

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

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

[60] Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 251–260, Washington, D.C., 28–30 May 1986.

[61] Dan 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.

[62] D.E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 114–121, San Francisco, California, 5–7 May 1982.