Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs

Robert Utterback* Kunal Agrawal* Jeremy T. Fineman† I-Ting Angelina Lee*

*Washington University in St. Louis
*{robert.utterback, kunal, angelee}@wustl.edu

†Georgetown University
†jfineman@cs.georgetown.edu

ABSTRACT

If a parallel program has ***determinacy race(s)***, different schedules can result in memory accesses that observe different values — various race-detection tools have been designed to find such bugs. A key component of race detectors is an algorithm for ***series-parallel (SP) maintenance***, which identifies whether two accesses are logically parallel.

This paper describes an asymptotically optimal algorithm, called ***WSP-Order***, for performing SP maintenance in programs with fork-join (or nested) parallelism. Given a fork-join program with T_1 work and T_∞ span, WSP-Order executes it while also maintaining SP relationships in $O(T_1/P + T_\infty)$ time on P processors, which is asymptotically optimal. At the heart of WSP-Order is a work-stealing scheduler designed specifically for SP maintenance.

We also implemented C-RACER, a race-detector based on WSP-Order within the Cilk Plus runtime system, and evaluated its performance on five benchmarks. Empirical results demonstrate that when run sequentially, it performs almost as well as previous best sequential race detectors. More importantly, when run in parallel, it achieves almost as much speedup as the original program without race-detection.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; E.1 [**Data Structures**]: *Distributed data structures*

Keywords

series-parallel maintenance, determinacy race, race detection, work stealing, order-maintenance data structures

1. INTRODUCTION

A ***determinacy race*** [14] (or a ***general race*** [29]), occurs when two or more logically parallel instructions access the same memory location, and at least one of the accesses is a write. Determinacy races can lead to nondeterministic program behaviors, and as such they are often bugs.[1]

Over the years, researchers have proposed several algorithms [26, 14, 31, 32, 5] for performing race detection "on the fly" as the program executes. These race detectors provide a fairly strong correctness guarantee — for a given input, they report a race if and only if the program contains a race on that input. In general, on-the-fly race detectors include two important components: (1) an ***SP-maintenance data structure*** for maintaining and querying series-parallel relationships between strands of a parallel program online; and (2) a ***memory access history*** that keeps track of (one or two) previous readers and writers for each memory location. On each memory access, the race detector checks whether the current access is logically parallel with these previous accesses to determine whether a race exists.

Since Mellor-Crummey's seminal work [26], most on-the-fly race detectors have adopted similar schemes for maintaining the access history. Where they differ significantly is in how they perform SP maintenance. Not only can SP maintenance significantly impact the overall running time of race detection, but it may dictate that the program (with race detection) must run sequentially. All prior SP-maintenance algorithms have performance drawbacks — either they must execute the program serially [14, 31], they limit the parallelism of the execution [5], or they have high worst-case overhead [26, 32] on memory accesses.

This paper describes ***WSP-Order***, an asymptotically optimal SP-maintenance algorithm for fork-join programs that runs in parallel. Specifically, let T_1 be the ***work***, or sequential running time, of a deterministic computation[2], and let T_∞ be its ***span***[3], or its running time on an infinite number of processors. Then WSP-Order executes the computation while correctly maintaining SP-relationships in expected time $O\left(\frac{T_1}{P} + T_\infty\right)$ on P processors. Since running the computation without SP maintenance also takes $\Omega(\frac{T_1}{P} + T_\infty)$ time, this is the best possible bound and guarantees linear speedup when the computation's parallelism is $\Omega(P)$.

Adding the standard access-history algorithm to WSP-Order, we get a correct and efficient parallel race-detection algorithm. The access history has some inherent overheads due to concurrent updates on reads; therefore, it is diffi-

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

 ISBN 978-1-4503-4210-0/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2935764.2935801

[1] In contrast, a ***data race*** occurs when the atomicity of critical sections is violated [29].

[2] Henceforth, we shall use the term ***computation*** to refer to a program given a fixed input.

[3] Span is sometimes called critical-path length or depth.

cult to provide good guarantees for race-detection on a weak contention model. Assuming a constant-time priority-write primitive [36], a race detector with WSP-Order executes a program with full race detection (including access history) in expected $O\left(\frac{T_1}{P} + T_\infty\right)$. To our knowledge, this is the first asymptotically optimal parallel race-detection algorithm.

SP Maintenance: WSP-Order achieves its running time through a nontrivial integration of a modified work-stealing scheduler [6] and a parallel version of SP-Order, an SP-maintenance algorithm proposed by Bender et al. [5]. At a high-level, SP-Order uses a pair of order-maintenance (OM) data structures [9, 4] to perform SP maintenance. When running sequentially, SP-Order is asymptotically optimal since OM data structures support constant-time operations. However, it is not immediately clear how to produce an efficient parallel version of SP-Order due to its reliance on the shared OM data structures. To compensate, Bender et al. [5] propose a significantly more complicated algorithm allowing parallel execution, but it has non-constant overhead.

WSP-Order achieves optimal performance without modifying the SP-order algorithm itself. Instead, we modify the scheduler and the OM data structures. A key insight is that SP-Order uses OM data structures in a well structured way — concurrent accesses to the data structures do not logically conflict; therefore, in principle, concurrent updates to the data structure can generally proceed without concurrency control. Occasionally, however, the OM data structures undergo ***relabel operations***, wherein a large portion of the data structure is modified. These relabels, which are necessary to achieve the optimal performance bounds, do conflict with concurrent operations. Coping with relabels in the OM data structure is the main challenge in parallelizing SP-order and achieving asymptotically optimal performance.

The high-level ideas behind WSP-Order are to (1) forbid concurrent accesses during relabels, but otherwise allow all concurrent OM operations to proceed without any significant concurrency control; (2) use parallelism within relabel operations to ensure that they can finish quickly; (3) modify the scheduler to prioritize relabels; and (4) ensure that relabels do not occur "too frequently" by relabeling more eagerly. As we shall see in later sections, these four modifications are sufficient to achieve provably good performance.

Race Detector: Based on WSP-Order, we implemented a race detector, ***C-RACER***, in Cilk Plus [20], a multithreaded language that supports fork-join parallelism. Empirical results on five benchmarks indicate that C-RACER outperforms an existing state-of-the-art race detector for Cilk Plus [21]. They bear out the claim that WSP-Order is asymptotically optimal — when executing on 16-cores, C-RACER achieves speedups similar to the computation without C-RACER. A breakdown of overheads shows that, the C-RACER's overheads are mostly incurred in access-history management, and the WSP-Order indeed incurs minimal overhead, less than 2× the baseline across benchmarks.

Outline: Section 2 provides background about work stealing, race detection, and OM data structures. Sections 3 and 4 describes the design and theoretical analysis of WSP-Order. Section 5 shows empirical results for C-RACER. Sections 6 and 7 provide related work and conclusions.

2. BACKGROUND

In this section, we start by describing the parallel control constructs for generating fork-join parallelism and the work-stealing scheduler. Next, we describe the abstract data type for the order-maintenance (OM) data structure and how SP-Order uses the it to perform SP-maintenance. We then review how a parallel race detector maintains an access history and detects races. Finally, we describe the order-maintenance data structures and their complexity.

Fork-Join Programs: We will use Cilk Plus keywords to explain the fork-join model; other languages have similar constructs. Parallelism is created using `cilk_spawn`. When a function instance F ***spawns*** another function instance G by preceding the invocation with `cilk_spawn`, the ***continuation*** of F — the statements after the spawning of G — may execute in parallel with G without waiting for G to return. Instruction `cilk_sync` acts as a local barrier; the control flow cannot move past a `cilk_sync` in function F until functions previously spawned by F have returned.

A parallel computation can be represented as a directed acyclic graph (dag). Each node in the dag is a ***strand*** — a sequence of instructions that contain no parallel primitives and therefore must execute sequentially. Nodes become ***ready*** to execute when all their predecessors have executed.

For fork-join computations, the dag is ***series parallel*** [14], meaning that it can be generated by repeated series and parallel composition. A race detector must maintain series-parallel relationships between strands. Formally, a strand x is ***logically precedes*** strand y if and only if there is a directed path from x to y in the computation dag. If there is no directed path in either direction, the two strands are ***logically parallel***.

Work-Stealing Scheduler: During execution, a ***work-stealing*** scheduler [6, 17] dynamically load balances a parallel computation across available ***worker*** threads. Each worker maintains a ***deque***, double-ended queue; when a worker creates new strands, they are placed on this worker's deque. When it completes its current strand, it takes work from the bottom of the deque. If its deque becomes empty, the worker becomes a ***thief*** and randomly chooses a ***victim*** worker to ***steal*** from. Given a computation with work T_1 and span T_∞, a work-stealing scheduler executes it in expected time $\frac{T_1}{P} + O(T_\infty)$ on P processors [6].

Order Maintenance Abstract Data Type: An ***order-maintenance (OM)*** data structure maintains a total order of elements subject to the following operations.

- PRECEDES(x,y): Given pointers to x and y, return *true* if x precedes y in the total order and *false* otherwise.
- INSERT(x,y_1,y_2): given a pointer to existing element x, splice-in new elements y_1 and y_2, in that order, *immediately* after x in the total order. Thus, x and all its predecessors of x precede y_1 and y_2, while all successors of x succeed y_1 and y_2.

It is possible to solve order maintenance in $O(1)$ time per operation[9, 4]. The amortized solutions are simpler (and sufficient for our purposes), but it is possible to achieve the same bounds in the worst case [9, 4].

The SP-Order Algorithm: SP-Order [5] uses two order-maintenance data structures to maintain two different total orderings — called ***English*** and ***Hebrew*** — of all strands in the computation. Figure 1 shows pseudocode for SP-Order.[4]

[4]The original description of SP-Order is with respect to a series-parallel parse tree; we adopt an operational description here.

In the frame of each function F, SP-Order has pointers to elements in English and Hebrew structures representing its currently executing strand. On spawns, it creates nodes for three new strands: the newly spawned function, the continuation strand, and the strand immediately after the corresponding `cilk_sync` (this last one need only be created for the first spawn of the sync block). These three new elements are inserted after the current strand in both orderings. In the English ordering their order is spawn then continuation then sync. In the Hebrew ordering, their order is continuation then spawn then sync. These orderings are sufficient to determine SP relationships; a strand x logically precedes strand y if and only if x precedes y in both orderings.

```
On F spawning G :
 1  if first-spawn = true
 2    first-spawn = false
 3    create OM-Element for F.sync.e and F.sync.h
 4    OM-Insert(Eng, F.curr.e, F.sync.e)
 5    OM-Insert(Heb, F.curr.h, F.sync.h)
 6  create OM-Element for G.curr.e, G.curr.h,
    F.cont.e, and F.cont.h
 7  OM-Insert(Eng, F.curr.e, G.curr.e, F.cont.e)
 8  OM-Insert(Heb, F.curr.h, F.cont.h, G.curr.h)
 9  F.curr.e = F.cont.e;
10  F.curr.h = F.cont.h

On F passing cilk_sync :
11  first-spawn = true
12  F.curr.e = F.sync.e
13  F.curr.h = F.sync.h

On F calling child H :
14  H.curr.e = F.curr.e
15  H.curr.h = F.curr.h

On a called child H returning to F :
16  F.curr.e = H.curr.e
17  F.curr.h = H.curr.h
```

Figure 1: The SP-Order algorithm. SP-Order maintains two OM data structures, *Eng* and *Heb*. For a function F, elements representing the currently executing strand are in $F.curr.e$ and $F.curr.h$. $F.cont.e$ and $F.cont.h$ represent F's continuation strand after the spawning of G. $F.sync.e$ and $F.sync.h$ represent the strand after the corresponding `cilk_sync` in F.

We use the SP-Order's SP-maintenance algorithm without modification; therefore, the correctness guarantee follows from correctness of SP-Order (presented in [5]). In addition, while SP-Order was described as sequential algorithm, it works in parallel out of the box if we can insert and query into the English and Hebrew structures concurrently. (The question is performance.)

Detecting Races Using SP-Order: The second main component of a race detector is the access history. WSP-Order maintains an access history, briefly overviewed here, similar to that used by Mellor-Crummey's [26] race detector. For a parallel race detector, it is sufficient for the access history to record the last writer w and two readers — the "left-most" reader lr and the "right-most" reader. When a strand s writes to ℓ, the race detector now performs three queries in the SP-maintenance structure, reporting a race if any s is logically parallel with any of lr, rr, or w. If not, s becomes the last writer. If s reads ℓ, s is only compared against w to find a race as before. However, additional queries are necessary to see if s is the new leftmost reader lr or rightmost reader rr. Since multiple reads can occur in parallel even in a race-free program, more than one reader could try to update these locations concurrently. Therefore, the parallel race detector needs to protect these objects using locks or atomic primitives like compare-and-swap or priority writes.

Overview of Order-Maintenance Implementations: Efficient implementations of order maintenance can provide $O(1)$ time per operation [9, 4] for both inserts and queries using two main insights. First, these use a ***two-level data structure***. Each element is stored in a ***bottom level structure*** which contains a ***group*** of consecutive elements. The groups (i.e., pointers to bottom-level structures) are organized into a single ***top-level structure***. Second, each element and each group has an integer label L_B and L_T, respectively; group labels are unique and element labels within a group are unique. The OM data structure maintains the invariant that element x precedes element y in the total order if and only if (1) $L_T(group(x)) < L_T(group(y))$, or (2) $group(x) = group(y)$ and $L_B(x) < L_B(y)$. Therefore, the Precedes query only requires two label comparisons.

An Insert(x,y) is implemented by trying to inserting y immediately after x in x's bottom-level structure and assigning y a unique label between x and x's successor in the group. If y can be assigned a label, then the insert completes in $O(1)$ time. If y cannot be assigned a label e.g., x's successor has label $L_B(x) + 1$, then the group is ***full***, triggering a ***relabel operation***. A relabel operation consists of two steps: (1) The full group is ***split*** into potentially many groups, the elements therein are assigned new labels that are "spaced apart" allowing for fast future inserts. (2) The (pointers to) the newly created groups are inserted into the top-level structure. During this step, the top level structure is potentially ***rebalanced*** and the group labels may be changed to accommodate these new groups. There are a few details and many different labeling schemes exist, but the important features are (1) the amortized cost of a split is $O(1)$ per insert, and (2) each group contains $\Omega(\lg n)$ elements, so the total number of splits and top-level insertions is $O(n/\lg n)$. The cost of inserting new groups into top-level structure is $O(\lg n)$ on average; combining these features, the amortized cost per insert is $O(1)$.

3. EFFICIENT PARALLEL SP-ORDER

This section discusses how to get an efficient, parallel implementation of SP-order by modifying both the work-stealing runtime system and the OM data structures. Here we present the theoretically optimal design; in our implementation (see Section 5), we slightly modified both the runtime system and the OM data structure for both ease of programming and practical performance.

3.1 Overview

Multiple challenges are involved in obtaining a theoretically efficient parallel SP-Order. First, SP-maintenance may perform *a lot* of OM operations since every spawn or sync translates into OM inserts and each memory access translates to multiple OM queries (see Section 2). In the worst case, the number of inserts and the number of queries can both be $\Theta(T_1)$, where T_1 is the work of the computation. Since all P processors can be continually pounding on the

```
INSERT(x,y) // executed by worker w
 1  y_inserted = false
 2  repeat
 3    g = GROUP(x)
 4    if IS_FULL(g)
 5      if TRY_ACQUIRE(global_lock)
 6        // push RELABEL onto DS deque and
 7        START_RELABEL() // switch to DS deque
 8      else
 9        JOIN_RELABEL() // switch to DS deque
10    elseif TRY_ACQUIRE(w.local_lock)
11      INSERT_INTO_GROUP(g,x,y)
12      y_inserted = true
13      if IS_HEAVY(g)
14        ADD_TO_HEAVY_GROUPS(g)
15      RELEASE_LOCK(w.local_lock)
16    else // local lock not acquired, relabel in progress
17      JOIN_RELABEL() // switch to DS deque
18  until y_inserted = true

RELABEL() // invariant: global lock is held
1  parallel for each worker w
2    ACQUIRE(w.local_lock) // spin until successful
3  Build array H of all heavy groups
4  parallel for each index i of H
5    PARALLEL_SPLIT(H[i])
6  PARALLEL_REBALANCE_TOPLEVEL()
7  parallel for each worker w
8    RELEASE(w.local_lock)
9  RELEASE_LOCK(global_lock)
```

Figure 2: Pseudocode for the insert and the relabel procedures of the OM data structure.

data structures, it is natural to expect an overhead of at least $\Omega(\lg P)$ to coordinate the accesses. Indeed, most concurrent data structures have such overheads (or worse). Nevertheless, our approach avoids this overhead.

A key idea of our approach is to bypass the expensive coordination for most OM operations (inserts and queries). Doing so seems impossible for a general concurrent data structure — after all, some coordination is required to guarantee correctness — but extra structural properties of both SP-Order and the order-maintenance data structures help us. Three key observations that lead to our good design:

1. In SP-Order, all OM insertions are with respect to the strand currently being executed by that worker. Thus, there are no logical conflicts between any concurrent insertions.
2. Barring splits due to relabels, inserting into a bottom-level structure requires only local changes. Consequently, as long as no splits are in progress, concurrent inserts may safely operate on disjoint locations in bottom-level structures.
3. Queries are just comparing labels and groups. As long as no labels or groups are changing, queries and inserts can occur concurrently.

Given these insights, the main idea is simple: resort to a slow concurrency-control mechanisms only during a relabeling.

Even with the reduced concurrency control, a second challenge remains: relabels can be large operations and potentially block inserts and queries. Executing all relabel operations sequentially would make it impossible to provide good speedup. Therefore, (1) we design an OM data structure that implements parallel relabels; and (2) we modify a work-stealing scheduler to prioritize relabels and properly move processors between working on the computation and the ongoing relabel operation as needed. The combination of the two strategies helps speed-up the relabel process.

Finally, the last challenge is that, with a traditional OM data structure, there may be as many as $\Theta(n/\lg n)$ relabel events over the lifetime of an n-element order-maintenance structure. Since a parallel relabel operation incurs $\Theta(\lg n)$ span and each relabel occurs one after another, all relabels together would incur $O(n)$ span, which can be as bad as $O(T_1)$. To allay this problem, we must somehow reduce the number and frequency of distinct relabel events. Recall that, during a relabel, a full group (as implemented by a bottom-level structure) is split into multiple groups. Our solution is to introduce a more aggressive process — once a relabel is triggered, instead of just splitting the full group, we also split all groups that are "heavy," or full enough. This aggressive splitting also introduces more parallelism into the relabel process, since multiple groups can be split in parallel.

In summary, the four main components of our solution are (1) reducing concurrency control except on relabels, (2) parallelizing the relabels, (3) modifying the work-stealing scheduler to prioritize relabels, and (4) reducing the frequency of distinct relabel operations by splitting more aggressively.

The remainder of this section explains each of these in detail. We first explain the mechanism for reducing concurrency control and the scheduler modifications. Then we describe the implementation of the OM data structure used by WSP-Order. Finally, we present how to perform fewer relabels and how to do relabels in parallel.

3.2 Reducing Concurrency Control

To reduce concurrency control, WSP-Order uses a two-level locking scheme. Each processor/worker has a ***local lock*** that it must acquire before performing any data-structure operation (insert or query). There is also a single ***global lock*** to protect relabel. When a relabel is in progress, the relabel process acquires the global lock as well as all the local locks. Therefore, a relabel cannot be concurrent with any insert or query; however, inserts and queries from different workers can proceed concurrently with each other.

Figure 2 shows the pseudocode for the underlying INSERT operation used by OM-INSERT. When a worker w is trying to insert element y after element x, it first checks to see if the group that x belongs to is already full (line 4). If so, w either triggers a relabel process (line 7) if one is not in progress (the global lock is free), or it joins the ongoing one (line 9). If the group is not full, w proceeds to acquire its local lock (line 10) and perform the insert (line 11). If the local lock is not free, a relabel is in progress, and w joins the relabel (line 17). If the local lock is free, the insert always succeeds since the group was not full before the insert. The insert may cause the group to become heavy or full, however, in which case the group is marked as heavy (line 14) and will be split in the next relabel. (We will describe the condition a group being heavy and the relabel process in later subsections.)

There are a few subtleties in the code shown. First, checking if the group is full must be in the loop, since even though w may join an ongoing relabel process, that relabel process may not include g (for instance, the relabel process started before g became heavy). In this case, g remains full after this particular relabel operation finishes, and w will need to recheck g for fullness and possibly start the another relabel

operation. Second, the local-lock acquire must also be in the loop, since w may fail to acquire the lock, join the relabel, and must retry the insert again when the relabel ends. Finally, since RELABEL is a parallel operation, the worker who initiated RELABEL's may not be the one who executes its final instruction. Therefore, it is important to release the global lock at the end of RELABEL (as opposed to in INSERT after START_RELABEL returns).[5]

3.3 Scheduler Support to Prioritize Relabels

Since an ongoing relabel prevents other concurrent OM operations, we want an ongoing relabel to finish quickly; moreover, workers blocked on an insert or query should help with the relabel instead of being idle. To prioritize relabels, we modify the work-stealing scheduler as follows. Each worker has two deques, a ***core deque*** and a ***DS deque***. The DS deque is only used to hold work associated with the parallel relabel, and the core deque holds all other work of the program. When workers steal work from some other worker's core deque, they place it on their own core deque, and when they steal work from some worker's DS deque, they place it on their own DS deque. Initially all workers perform work stealing on core deques as usual, switching to the DS deque when starting a parallel relabel operation.

In general, workers prioritize the DS deque, enabling relabels to finishes quickly. In particular, when a worker w starts a relabel (START_RELABEL in line 7), w suspends the current strand, switches to the DS deque to work on relabel, and does not return until discovering that the relabel has finished. Likewise, when a worker w fails to acquire a global or local lock (i.e., a relabel is in progress), w invokes JOIN_RELABEL (in lines 9 and 17), which suspends the current strand, switches to work-steal on the DS deque until the relabel finishes. Finally, when the current deque is empty, w checks the global lock before performing a steal attempt. If the global lock is held, it starts work stealing on the DS deques. If the global lock is free, it operates on the core deque, resuming any suspended strand. If the core deque is empty, it starts work stealing from core deque.

3.4 The OM Data Structure Implementation

Before we describe the parallel relabel and how we reduce its frequency, we must first describe in more detail the implementation of the OM data structure used in WSP-Order.

Bottom-level structure: In sequential versions of OM data structure, the bottom-level structures are typically implemented as linked lists. WSP-Order, however, must iterate over all elements in parallel during relabels. Therefore, in WSP-Order, these are implemented as unbalanced binary tree where each internal node has exactly two children. All the OM elements are stored in the leaves. Each element has an integer label corresponding to its root-to-leaf path (as in a trie), i.e., 0 for left child and 1 for right child, with the root starting from the high-order bit of a b-bit machine word. Any unused trailing bits are implicitly 0s. Each element also stores its depth in the tree (i.e., the logical label length) and its group identifier.

To insert a new element y after an existing element x, first assign y the same group as x. Next, create a new internal node with x and y as the left and right children, respectively, and splice that node in place of x in the tree. Next, assign y the label induced by its root-to-leaf path by appending a 1 to x's label, i.e., $L_B(y) = L_B(x) + 2^{b-depth(x)}$. Finally, record y's depth and implicitly refine the precision of x's label by incrementing x's depth. It is not hard to see that y's label correctly matches the root-to-leaf path to y and that these updates can be performed in constant time. Note that the label induced by x's path does not change since 0s are appended regardless, so concurrent queries on x are not affected by the update.

The structure is considered ***full*** when the depth of any node reaches b — in this case, we no longer have space to insert another node immediately after the currently inserted node while keeping the number of bits in the label at most b. At this point, a relabel is triggered. During a relabel, all the full bottom level structures are ***split*** — that is, all the elements from a full bottom-level structure are partitioned between multiple structures of smaller depth. This leads to a change in labels of all these elements.

Top-level structure: WSP-Order needs to operate on the top-level structure in parallel, so a structure like Dietz's original tree-based version [10] is most appropriate. Here, we describe the basic sequential structure for concreteness, and while some of the details differ slightly from previously published structures, we are not claiming any new ideas here.

The top-level structure is a balanced binary tree where each internal node has exactly two children. Like the bottom-level structures, all elements are stored in the leaves, and each element is explicitly labeled according to its root-to-leaf path. The elements (leaves) of the top-level structure are (pointers to) the bottom-level groups. Since the top-level structure can be much larger than bottom-level structures, we must maintain a balanced tree to ensure that labels fit in a machine word. Therefore, the easy insertion algorithm described earlier is not sufficient. On the other hand, all inserts into the top-level structure occur inside relabel operations, so they are protected by the global lock.

Since the labels of elements change whenever its root-to-leaf path changes, rotation-based balancing schemes do not readily apply. Instead, most implementations of the top-level structure use some form of weight balance. For each node, WSP-Order maintains a ***size*** corresponding to the number of leaves in the node's subtree, and a ***depth*** corresponding to the distance from the root to the node. We say that a node r is ***out of balance*** if some specific conditions on $size(r)$ are not met. When a node becomes out of balance, the entire subtree rooted at r must be rebuilt. It is not important to understand the specific condition used — there are many balance conditions that would work, e.g., scapegoat trees [3, 18]. Rebuilding the appropriate subtree can be easily implemented as parallel tree walks, with $O(size(r))$ work, which is sufficient to achieve $O(\log n)$ work per top-level insert [3, 18].

3.5 Reducing Relabel Frequency

WSP-Order reduces the frequency of relabels by proactively splitting any bottom-level structures that are "not full, but close enough" whenever a relabel occurs. In particular, we say that a bottom-level structure is ***heavy*** when its depth reaches $b/2$, where b is the number of bits in a word. This small change adds much-needed parallelism to relabels by allowing multiple splits to occur during a single relabel.

We make two changes to keep track of heavy lists. First, each bottom-level structure has a bit flag to indicate whether

[5]Our locking implementation allows a lock to be acquired and released by different workers.

the structure is heavy. Second, each worker has a local array that stores a list of heavy groups to be read when performing a relabel. When a worker w performs an insertion that results in a $\geq b/2$-bit label, w performs a test-and-test-and-set on the heavy bit. If w manages to set the bit, then a pointer to this group is added to w's local array of heavy groups; otherwise, w does not retry. This procedure ensures that a heavy group is stored on only one worker's local array and trivially takes $O(1)$ time.

3.6 Parallel Relabels

We now describe how to parallelize the relabel operation. Recall that a relabel operation splits each the heavy groups (bottom-level structures) into potentially many groups and inserts elements representing newly creating groups into the top-level structure, also rebalancing the top-level structure.

Figure 2 also shows the high-level control flow for the RELABEL procedure. It first acquires all local locks in parallel. These lock acquisitions are blocking, retrying until successful. Second, all heavy lists from P workers' local arrays are concatenated in parallel using prefix sums (line 3). Next, in parallel, we split all heavy groups. Since groups may be large, each split is further parallelized using two parallel, recursive tree walks. The first walk calculates the size of the subtree of each internal node. (These calculations are performed in postorder, i.e., when returning from recursive tree walks.) Given a size-m bottom-level tree, the second walk splits the tree into $\Theta(m/b)$ shallow, fully balanced bottom-level trees of size $\Theta(b)$ and depth $\Theta(\lg b)$. One method is to write all elements into an array in order (in parallel) using size information at internal nodes to determine array offsets, split the array into subarrays of $\Theta(b)$ elements, and then recursively build shallow balanced trees of these segments. Pointers to these new groups are stored in an array attached to the original group's node, for easy parallel access.

Rebalancing the top-level tree in parallel is more complicated. Since we split the heavy groups into multiple groups, some leaves in the top level tree (corresponding to groups that were just split) now represent multiple groups and the ancestors of these nodes have a different size now. We must be careful when updating sizes to avoid race conditions on the internal nodes. The process is as follows:

1. For each split group in parallel (i.e., a parallel loop over the list of heavy groups), follow the leaf-to-root path up the tree, marking all ancestor nodes via test-and-set (no locks needed because the races are benign).
2. Recursively walk down the tree in parallel, only along the marked paths, updating the sizes of internal nodes in postorder as before. For the base case, the size of each modified leaf is the number of new bottom-level structures into which that group was split.
3. Recursively walk down the tree again in parallel, stopping the recursion at each branch that finds an out-of-balance node. Note that there may be many out of balance nodes, but starting from the root finds all the highest ones.
4. Rebuild all out-of-balance subtrees in parallel. Similar to bottom-level tree splits, we can first recursively build an array of leaves, and then use this array to construct a new balanced subtree.

Finally, all marked nodes are unmarked in parallel, all heavy arrays emptied in parallel, and then all local locks released in parallel before releasing the global lock.

4. PERFORMANCE ANALYSIS

This section analyzes the running time of a program augmented with WSP-Order to perform SP-maintenance. Our analysis divides the execution into phases. A ***core phase*** is a period during which only core work (the original computation) and inserts and queries to OM data structures occur, but no relabels. A core phase ends when a relabel operation begins, at which point a ***relabel phase*** begins; it ends when the relabel finishes and the next core phase begins.

As long as inserts and queries (apart from relabel) take $O(1)$ time, it is not hard to see that the total time spent in core phases is $T_P = O(T_1/P + T_\infty)$ in expectation for a program with T_1 work and T_∞ span on P processors, following from a standard work stealing analysis [6]. The challenge is stitching together the core phases and the relabel phases. The crux of the argument is to leverage the fact that relabels are provably infrequent, and hence the number of relabel phases is $O(T_P/\log n)$. We then apply a work-stealing analysis to each relabel phase to conclude that they also take $O(T_1/P + T_\infty)$ time overall.

4.1 Performance Model

For our theoretical analysis, we make the following modeling assumptions. These assumptions are common in the related literature but not always called out explicitly.

First, our base model is a transdichotomous RAM [16], meaning that (1) b-bit machine words support standard arithmetic and bit operations in constant time, and (2) all pointers fit in a single word. Therefore $b > \lg n$, and all top-level labels fit in a constant number of machine words. Similarly, $b > \lg P$. Second, we assume that all processors operate at the same speed, e.g., that workers are not swapped out to run an operating-system task. Third, for SP maintenance and work stealing, the basic atomic primitive is a compare-and-swap (CAS). We assume that each CAS completes (possibly failing) in constant time regardless of contention. Finally, we assume that lock (which can be implemented using CAS9) are somewhat fair. That is, if worker w_1 is spinning on a lock, and worker w_2 releases the lock, then we assume that w_1 acquires the lock before w_2 can reacquire it. In WSP-Order, there is never more than one worker spinning on a lock, so this assumption is reasonable. All these assumptions are for performance only, not correctness.

4.2 Core Phases

We argue that OM operations during core phases are fast and bound the total time in core phases.

LEMMA 4.1. *Each data-structure operation performed during a core phase completes in $O(1)$ time and does not conflict with any other operations.*

PROOF. By definition, during the core phase the global lock is not held, and no relabels occur. Thus, an insert operation consists of 1) acquiring the processor's local lock, 2) performing an insert into a bottom-level structure, 3) potentially performing a test-and-set on a group and adding a heavy group to the processor local array of heavy groups, and 4) releasing the local lock. Since there is no competition on local locks when the global lock is not held and test-and-sets are constant-time, each step takes $O(1)$ time. All inserts add new elements after the element representing currently executing strand in the OM structures. Since each strand is executed by a different processor, concurrent inserts operate

with respect to different insertion points and there are no conflicts during inserts. Queries are similar, modulo the details of the query itself which also takes $O(1)$ time. To see that inserts do not conflict with queries, it is enough to observe that 1) all queries by SP-order query elements that have already been inserted into the OM structures, and 2) in the absence of relabels, inserts do not change labels of any existing elements. □

LEMMA 4.2. *Consider a P-processor execution of a series-parallel computation augmented to perform SP-maintenance. Let T_1 and T_∞, respectively, denote the work and span of the underlying* a posteriori *computation DAG not counting the SP-maintenance operations. Then the total time spent in core phases, including SP-maintenance operations, is $T_P = O(T_1/P + T_\infty)$ in expectation.*[6]

PROOF. First, create a ***augmented core DAG*** by augmenting every node in the computation DAG by $O(1)$ to reflect the worst-case time of any OM operation apart from relabels (with constant chosen according to Lemma 4.1).

Observe that, using standard work-stealing [6], the augmented core DAG executes in $O(T_1/P + T_\infty)$ expected time on P processors. It remains only to confirm that the execution of core phases is no worse than executing the augmented core DAG. In particular, consider the execution with all relabel phases elided. By Lemma 4.1, all data-structure operations during this subexecution take $O(1)$ time. Moreover, when a processor starts working on its DS deque (only during a relabel phase), it simply suspends works on its core deque and resumes where it left off at the start of the next core phase. The execution thus corresponds to a work-stealing execution of the augmented core DAG, except that some nodes (finished during relabel phases) may be removed for free; specifically some processors (those neither stealing nor performing data-structure operations) may continue to make progress on their core deques during relabel phases, which only improves the time bound of core phases. □

4.3 Relabel Phases

We next analyze the work and span of relabel phases, which we can use to get total time spent in relabel phases by applying a work-stealing analysis. We will subdivide the phases further to consider the running time.

LEMMA 4.3. *The amortized work complexity of each order-maintenance insert is $O(1)$.*

PROOF. This analysis is very similar to the standard order-maintenance analyses, e.g., [9]. The only difference is in our parallel implementations of the structures.

Consider any bottom-level tree that is split, i.e., that is heavy at the start of the relabel phase, and m be the number of elements (leaves) in the tree. The split is implemented as several (parallel) tree walks, so the work is asymptotically the number of nodes in the tree. Since each internal node has 2 children, the number of internal nodes is less than m. The total work is thus $O(m)$.

To get $O(1)$ work per element, we need to charge this $O(m)$ work against $\Omega(m)$ new insertions. Again consider the bottom-level tree. When this tree was created (the result of a previous split), it had $m' = \Theta(b)$ elements, and it was as balanced as possible, i.e., having depth $\lg b + O(1)$. Since it is now heavy, its depth has grown to $b/2$, meaning that it has experienced at least $\Omega(b)$ inserts. We charge the cost of the split against these insertions. That is, we divide the $O(m)$ cost by the $m - m'$ new insertions. Since $m' = O(b)$ and $m - m' = \Omega(b)$, we conclude that $m - m' = \Omega(m)$. The work per insertion is thus $O(m/(m - m')) = O(1)$.

The analysis of rebalancing the top-level structure is similar to that of scapegoat trees [18], where the amortized work of an insert is $O(\lg n)$ as long as the cost of rebuilding a subtree is linear in its size. In WSP-Order, the parallel rebalance process has two components 1) following leaf-to-root paths for each top-level insert, giving a worst-case cost of $O(\lg n)$ per top-level insert, and 2) tree walks, giving cost linear in size of the subtrees walked, which yields $O(\lg n)$ amortized work per the standard analysis. Adding these gives $O(\lg n)$ amortized work per top-level insert. Since each bottom-level structure contains $\Omega(b)$ elements, the number of top-level inserts is $O(n/b)$ and giving the cost per insertion of $O((1/b)\lg n) = O((1/\lg n)\lg n) = O(1)$. □

LEMMA 4.4. *The span of each relabel is $O(b)$, where b is the number of bits in a machine word.*

PROOF. The algorithm first acquires all P local locks. Since local locks are fair, it never has to wait for more than 1 operation to finish before it gets a lock and all non-relabel operations finish in $O(1)$ time. Therefore, it can acquire all local locks in parallel with $O(\lg P)$ span. Concatenating each worker's local list of heavy nodes and then splitting these list in parallel can be done with span $O(\lg P + \lg n)$. Next, bottom-level splits take $O(b)$ to perform divide and conquer on the trees. Finally, the top-level rebuilding has $O(\lg n)$ span to walk up the tree and $O(\lg n)$ span to perform the rebuilding tree walks. Adding these up gives $O(\lg P + \lg n + b) = O(b)$ by the transdichotomous assumption. □

To bound the time in relabel phases, we subdivide the phases further. A ***busy subphase*** occurs while at least $P/2$ processors are still working on the core deques, i.e., since the beginning of this relabel phase, they have neither stolen nor tried to acquire a the local lock to perform data-structure operations. The remainder of the relabeling phase is the ***stealing subphase***, i.e., when most processors are either working off or trying to steal from DS deques. Note that each relabel phase has at most one busy subphase and one stealing subphase — the busy subphase starts at the beginning of the relabel phase, and the stealing subphase persists to the end. But it is possible for a relabel phase to be entirely a busy subphase or entirely a stealing subphase.

LEMMA 4.5. *Consider a relabel phase consisting of a w-work relabel. The expected time spent in the stealing subphase is $O(w/P + b)$, where b is the size of a machine word.*

PROOF. By definition of a stealing phase, at least $P/2$ processors are performing work stealing on the DS deque. We can thus apply the work-stealing theorems to a dag with $w+P$ work and b span (Lemma 4.4), where the P work comes from the work of acquiring/releasing all P local locks. This gives expected time $O((w + P)/P + b) = O(w/P + b)$. □

[6] In fact, this bound can be extended to a high probability bound as in [6]. Specifically, with probability at least $1 - \epsilon$, the time is $O(T_1/P + T_\infty + \lg P + \lg(1/\epsilon))$.

4.4 Total Time across Relabel Phases

Consider the time bound in Lemma 4.5. The work term, $O(w/P)$, is good — the work is low (according to Lemma 4.3) and the term implies linear speedup. The problem is the span term of $O(b)$. Specifically, if there are too many relabel phases, e.g., n/b of them, then all we have concluded is that the total time spent in relabel phases is $O(n)$, and we would be better off just running sequentially.

To get a good bound on runtime, we argue next there are not too many relabel phases. We do so by arguing that core phases have to be long.

LEMMA 4.6. *Let T_P be the total time spent in core phases. Then there are at most $O(T_P/b)$ relabel phases.*

PROOF. In 1 timestep of a core phase, each worker can perform at most one insert into the bottom-level structures. Since each insert is to a different node in the tree, the depth of a bottom-level tree increases by at most 1 per timestep.

In addition, at the start of a core phase, all bottom-level trees have depth at most $b/2$. In particular, any trees deeper than $b/2$ are marked as heavy and split during a relabel phase. When split, any new bottom-level trees are built with a height of $\lg b + O(1) < b/2$ for sufficiently large b. Since, by definition, a core phase ends when some bottom-level tree reaches depth b, each core phase requires at least $b/2$ time. It follows that the number of core phases (and hence the number of relabel phases) is at most $O(T_P/b)$ □

We are now ready to sum the time across all relabel phases. We consider the busy and stealing subphases separately.

LEMMA 4.7. *The total time in busy subphases is $O(T_1/P + T_P/b)$, where T_1 is the work of the computation DAG not counting data-structure operations, P is the number of processors, and T_P is the total time of the core phases.*

PROOF. By definition, a busy phase is the time during which at least $P/2$ processors are working on the main computation. Ignoring failed lock acquisitions, each step makes $\Omega(P)$ progress towards the work of the augmented core DAG. Unfortunately, some steps may be unproductive, namely when the local lock acquisition fails. But this can only happen once per processor per relabel phase, after which point the worker shifts to the DS deque. Moreover, all attempts to acquire the lock take $O(1)$ time. The total work, including failed lock acquisitions, is $O(T_1 + PT_P/b)$, where the T_P/b is the number of phases from Lemma 4.6. Dividing by the $P/2$ yields the claim. □

LEMMA 4.8. *The total expected time in stealing subphases is $O(n/P + T_P)$, where n is the number of strands in the computation DAG, P is the number of processors, and T_P is the total time of the core phases.*

PROOF. Applying Lemma 4.5 across relabel phases, the total time across all k stealing phases is $O\left(\sum_{i=1}^{k}(w_i/P + b)\right)$, where w_i is the work of the ith stealing phase. By Lemma 4.3, $\sum_i w_i = O(n)$. By Lemma 4.6, the number of phases is $k = O(T_P/b)$. We're thus left with $O\left(\sum_{i=1}^{k}(w_i/P) + \sum_{i=1}^{k} b\right) = O(n/P + (T_P/b)b)$, which proves the claim. □

4.5 Total Time

Finally, we get the overall running time by adding the time for core phases and relabel phases together.

THEOREM 4.9. *Consider P-processor execution of a series-parallel computation augmented with WSP-Order. Let T_1 and T_∞, respectively, denote the work and span of the underlying* a posteriori *computation DAG, not counting the SP-maintenance operations. The completion time, including SP-maintenance operations, is $O\left(\frac{T_1}{P} + T_\infty\right)$ in expectation.*

PROOF. From Lemma 4.2, the total time in core phases is $T_P = O(T_1/P + T_\infty)$ in expectation. Adding the time in busy and relabel subphases as given in Lemmas 4.7 and 4.8, we get a total time of $O(T_P + (T_1/P + T_p/b) + (n/P + T_P)) = O(T_1/P + T_\infty)$ since $n = O(T_1)$. □

Since executing the computation without maintaining series-parallel relationships also takes $\Omega(\frac{T_1}{P} + T_\infty)$ time, this bound is asymptotically optimal, in expectation.

4.6 Performance of Race Detection

As described in Section 2, in addition to SP-maintenance, a parallel race detector must also maintain an access history which stores one writer w and two readers, lr and rr for each memory location ℓ. When a strand s reads (or writes to) ℓ, it uses the OM data structure to check if s is in parallel with w (or lr or rr) and reports a race if it is. Since each query in SP-Order takes $O(1)$ time, this race-detection operation takes $O(1)$ time per memory access.

There is, however, one subtlety. Strands also update the access history if certain conditions are met, recording their identities. Since concurrent writers only occur in the presence of a race, contention on the last writer is not a performance problem. The readers, however, are another story. Multiple strands reading ℓ simultaneously may be "left of" ℓ's current leftmost reader lr. This means that we need some concurrency control on the access history to avoid losing important updates. Note that this is an issue in all race detectors that run in parallel.

Fortunately, the "left of" (and "right of") relations induce total orders, and only the "leftmost" ("rightmost") reader's update need be performed. One can implement these relations using a priority-write primitive [36]. If priority updates complete in $O(1)$ time, which seems to match performance when the number of locations being updated is large [36], access history updates only adds $O(1)$ work per read. Under that assumption, our race detection algorithm, including the access history, runs in $O(T_1/P + T_\infty)$ expected time.

5. EMPIRICAL EVALUATION

This section evaluates the practical performance of a parallel race detector, C-RACER, that employs WSP-Order for SP-maintenance. We briefly overview the implementation of C-RACER and present experimental results that evaluate C-RACER's overhead and scalability. The results show that a program augmented with WSP-Order (but not access history) generally runs less than 2× slower compared with the baseline running without race detection. Full C-RACER also scales well, tracking the speedups obtained by the baseline and outperforming ***Cilkscreen*** [21], a well-engineered, state-of-the-art serial race detector from Intel when running on two or more cores. Finally, we also measured the frequency of relabel operations, and the results indicate that relabels occur infrequently and the eager splitting of heavy groups can indeed help.

Overview of Implementation: Our parallel race detector, C-RACER, contains multiple components: (1) WSP-

Order including a modified work-stealing runtime system as described in Section 3.3 as well as an implementation of the OM data structure that does parallel relabels and eagerly splits heavy groups; (2) compiler instrumentation for Cilk Plus's parallel constructs and memory accesses; and (3) an implementation of the access history. WSP-Order is based on an implementation of BATCHER [1], which was originally developed by modifying MIT Cilk [17], and we have ported it to Cilk Plus by modifying the Cilk Plus runtime. The instrumentation resembles previous work on tool annotations for Cilk Plus [38]; we have used the instrumentation developed for Cilkprof [34], which is implemented in a branch of LLVM/Clang compiler that contains Cilk Plus language extensions [19]. This branch of LLVM/Clang also implements ThreadSanitizer [35], which provides instrumentation for memory accesses. C-RACER's access history is maintained with a word (four-byte) granularity.

The actual implementation of C-RACER differs from the theoretical algorithm presented in Section 3 in a few ways. First, in the theoretical algorithm, if the global lock is held, workers always join a relabel operation when they steal. In our implementation, a stealing worker joins a relabel in progress with 50% probability. This scheduling strategy provides the same theoretical guarantees, but the full proof requires an analysis similar to BATCHER [1]. Second, we have implemented the bottom-level groups in the OM data structure as linked lists rather than trees, and the way we assign labels and marking lists heavy treats the lists as flattened trees. Thus the splits of the bottom-level groups are sequential. Third, instead of performing several traversals of the upper-level tree to determine size information, we walk up the tree only once, using atomic fetch-and-add instructions to increment the size of each node. Fourth, as mentioned in Section 2, parallel readers may update the access history for a given location concurrently, and the theoretical algorithm assumes the use of priority write [36]. In practice, we simply acquire a lock whenever we need to update the location of in the access history, with separate locks for the last writer, left-most reader, and right-most reader per memory location. The last three modifications potentially reduce the theoretical parallelism of the computation. However, they improve the overall performance due to lower overheads.

Experimental Setup: We used five benchmarks from the Cilk-5 distribution [17] to evaluate C-RACER. `matmul` performs divide-and-conquer matrix multiplication, `cilksort` runs a parallel mergesort, `fft` calculates a fast-fourier transform, `heat` performs heat diffusion, and `cholesky` calculates the Cholesky decomposition of a matrix.

We ran our experiments on an Intel Xeon E5-2665 with 16 2.40-GHz cores on two sockets; 64 GB of DRAM; two 20 MB L3 caches, each shared among 8 cores; and private L2- and L1-caches of sizes 2 MB and 512 KB, respectively. Running time are in seconds as a mean of five runs. The standard deviation was within 2% of the mean for most configurations, with a maximum of 8% for `cholesky`.

5.1 Overhead and Scalability

To evaluate the overhead of C-RACER, we compare the following configurations:

- **baseline:** execution without race detection;
- **WSP-Order:** execution augmented with WSP-Order but not the access history management.
- **full:** execution running C-RACER with full race detection (i.e., including both WSP-Order and the access history).

We obtain the WSP-Order configuration of C-RACER by turning on only the instrumentation for Cilk Plus's parallel control constructs but not memory accesses, allowing us to measure the overhead of only SP-maintenance (including relabel operations), but not include overheads due to access history management.

We also compare with the execution times of these benchmarks running with ***Cilksan***, a serial race detector that implements the SP-bags algorithm [14] for SP-maintenance, and with ***Cilkscreen*** [21], a well-engineered, state-of-the-art serial race detector from Intel that implements the SP-bags algorithm but instrumented using PIN [25], a binary instrumentation framework from Intel, instead of compiler instrumentation. We have implemented Cilksan for fair comparison, since unlike Cilkscreen, Cilksan uses the same compiler instrumentation and access history management as C-RACER; the only difference between Cilksan and C-RACER is the SP-maintenance algorithm used.

Table 1 presents running times of the different configurations. First, note that the overheads due to WSP-Order (SP-maintenance only) shows minimal overhead compared with the baseline. The benchmark with the highest overhead is `fft`, with 1.78 times overhead. This indicates that WSP-Order can be implemented efficiently. Most of the C-RACER overhead actually comes from the management of access history — full C-RACER incurs another 30–132× overhead, with `heat` being the worst case. This is expected because the WSP-Order performs additional work only at parallel control constructs and function boundaries, whereas the access history management incurs additional overhead on every memory access, and for most applications, the number of memory accesses far exceeds the number of `cilk_spawn`, `cilk_sync`, and function boundaries. The overhead for `heat` is particularly bad, for C-RACER as well as Cilksan and Cilkscreen, since compute-to-memory-access ratio is low.

Across all benchmarks, Cilksan outperforms Cilkscreen, so we compare to Cilksan. The ratio of C-RACER execution time to Cilksan's is generally less than 1.5. As column `full(16)` shows, however, C-RACER utilizes parallelism to significantly outperform Cilksan when running on 16 cores. In fact, as long as the application has some parallelism, C-RACER outperforms Cilksan on 2 or more cores.

To evaluate how well WSP-Order and C-RACER scale, Table 2 shows the speedup for each benchmark as we vary the number of cores used. For most benchmarks, our implementation of WSP-Order scales with the baseline, verifying the theory presented in sections 3 and 4. The exception is `cholesky`, which executes many spawns but only a small amount of work in each strand. In addition, the C-RACER also scales as the number of processors increases, generally at the same rate as the baseline program.

5.2 Detailed Breakdown of Overhead

Figure 3 shows a breakdown of the overheads due to various components of C-RACER for the `fft` benchmark. Due to lack of space, we present this breakdown only for `fft`. We chose `fft` due to the relatively high number of inserts and relabels required, which allows us to show meaningful results here. The x-axis shows the number of cores used, and the y-axis shows the aggregate processing time (i.e., number of cores used × wall-clock time) in seconds. As

	base	WSP-Order		full(1)		full(16)		Cilksan		Cilkscreen	
matmul	15.69	16.22	(1.03×)	499.61	(31.84×)	31.79	(2.03×)	408.49	(26.04×)	922.23	(58.78×)
cilksort	3.20	3.38	(1.06×)	113.29	(35.40×)	10.81	(3.38×)	73.71	(23.03×)	99.82	(31.19×)
fft	18.34	32.73	(1.78×)	878.17	(47.88×)	81.34	(4.44×)	523.94	(28.57×)	991.29	(54.05×)
heat	7.33	7.73	(1.05×)	1026.70	(140.07×)	79.55	(10.85×)	726.89	(99.17×)	1202.29	(164.02×)
cholesky	5.96	7.44	(1.25×)	687.88	(115.42×)	45.45	(7.63×)	666.66	(111.86×)	962.34	(161.47×)

Table 1: Execution times for five benchmarks, in seconds. The `full(16)` column shows the execution time of the full configuration of C-RACER running on 16 cores. All other columns show the execution time on a single core. The numbers in parenthesis indicates the overhead comparing with the baseline running serially.

		P				
		2	4	8	12	16
matmul	base	2.00	3.99	7.97	11.88	15.64
	WSP-Order	1.98	3.95	7.87	11.68	15.56
	full	2.00	3.98	7.95	11.82	15.71
cilksort	base	1.99	3.98	7.81	11.17	13.37
	WSP-Order	1.98	3.94	7.71	10.82	13.31
	full	1.92	3.51	6.25	8.19	10.48
fft	base	1.77	3.43	6.12	8.16	9.60
	WSP-Order	1.80	3.31	6.06	8.20	10.00
	full	1.71	3.27	6.43	8.81	10.80
heat	base	1.99	3.63	5.22	7.09	6.71
	WSP-Order	1.98	3.56	5.44	6.62	7.33
	full	2.06	3.86	7.43	10.71	12.91
cholesky	base	1.99	3.96	7.83	11.31	14.69
	WSP-Order	1.86	3.54	6.65	8.73	10.28
	full	1.98	3.87	7.64	11.44	15.13

Table 2: Speedup over the sequential version when running with different configurations. For each configuration, the speedup is computed with respect to the running time of the same configuration running on one core.

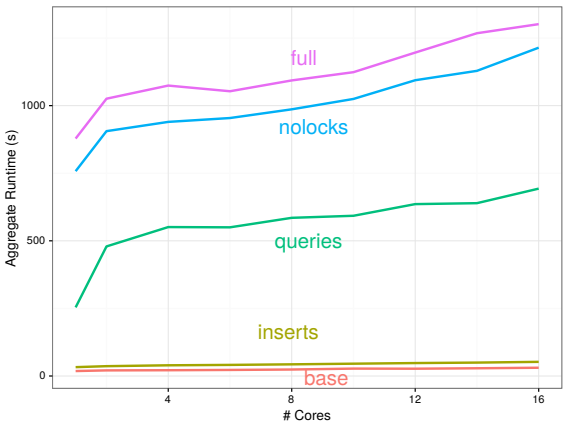

Figure 3: Detailed breakdown of C-RACER overheads for fft, in seconds. The lines `base` and `inserts` show the overall processing time when running the baseline and inserts configuration described earlier. The line `queries` shows the processing time after including the overhead for queries on top of the inserts configuration. The line `nolocks` shows the processing time of C-RACER running in full configuration but does not acquire locks when updating the shadow memory. The line `full` shows the overall processing time of C-RACER running in the full configuration.

the WSP-Order line shows, access history contributes the bulk of the overhead. The other three lines break down the overheads of various components of maintaining the access history. As explained in Section 2, the access history records three strands per memory location and on every access, the race detector queries both OM data structures with the current strands and all three strands, updating the access history as necessary. As shown in breakdown, half of the overhead of access history management is due to the 6 queries and the other half is due to checking and updating the access history. Moreover, whenever a strand s tries to update the shadow memory, it must acquire a lock. This locking overhead (the gap between lines `nolocks` and `full`) appears to be relatively small.

5.3 Effect of Eagerly Splitting Heavy Groups

We now analyze the effect of our modification to OM data structure where instead of only splitting full bottom-level groups, we eagerly split heavy groups. Recall that a tree is full when its depth reaches b and we set the heavy threshold as 50%; that is, a tree is considered heavy when it has depth $b/2$. In practice, we have implemented the bottom-level groups using lists, which means that we mark a list heavy when we discover a label with $b/2$ bits used before trailing 0s. We ask a few different questions here: (1) Does the eager splitting reduce the total number of relabels? (2) How many more splits do we get due to eagerly splitting trees? (3) What effect does the heavy threshold have on the number of relabels and the number of splits?

Table 3 shows the stats of relabel operations for three benchmarks running with C-RACER on 16 cores. These numbers are from a single run; general trends are similar across runs. We varied the heavy threshold (described here as the percentage of the total bits in a label — 64 in our case). For the most part, relabels occur infrequently; in fact, we omit the results for `matmul` and `cilksort` because their executions contain no relabels even for very large input sizes. For `heat` and `cholesky`, some relabels occur but not many. We see that the total number of heavy groups decreases as we increase the heavy threshold, but the number of relabels does not change. This makes sense, since with a lower threshold, more groups get marked as heavy early, but no relabel is triggered until some group becomes full.

The interesting one is `fft`, which triggers the most relabels among all our benchmarks. When the heavy threshold is 1 (last row) — that is, we only split full nodes — as expected, most relabels split only a single heavy group; very occasionally, a couple of heavy groups may become full concurrently and causes a relabel to split more than one group. Even by just decreasing the threshold to be .875, we already see that the median of the number of heavy groups increase to 5.7 and the number of relabels reduces dramatically. This trend continues as we decrease the heavy threshold. This result demonstrates that when many groups can become full dur-

	fft				heat				cholesky			
threshold	relabels	median	max	total	relabels	median	max	total	relabels	median	max	total
0.25	1567	18	40	32207	24	3	4	54	22	4	5	66
0.375	1576	15	27	21905	24	1	1	24	22	2	4	56
0.5	1614	19	22	20270	24	1	1	24	22	1	2	31
0.625	1642	12	22	18959	24	1	1	24	22	1	2	31
0.75	1725	7	17	17345	24	1	1	24	22	1	2	24
0.875	2838	4	11	11285	24	1	1	24	22	1	1	22
1	7985	1	3	8090	24	1	1	24	22	1	1	22

Table 3: The effects of varying the heavy threshold when running C-RACER 16 cores. The first column for a given benchmark shows the number of relabels. The other columns provide information about the number of heavy groups — the median and maximum for individual relabels, and the total number of heavy groups split across all relabels.

ing the execution, eager splitting indeed helps in decreasing the number of relabels.

6. RELATED WORK

Netzer [29] formalized definitions for determinacy and data races. Static checking for races has been studied in [12, 27, 13], while [8, 28, 2] investigate post-mortem analysis. Research on race detectors for general parallel computations include [15, 33, 30, 11]. In addition to English-Hebrew labeling and SP-Order (and the related SP-hybrid), many algorithms have been proposed for on-the-fly race detection for series-parallel programs. Offset-span labels[26] are shorter than English-Hebrew labels, but still not bounded by a constant. The Nondeterminator race detector [14] runs serially and uses Tarjan's nearly linear-time least-common-ancestor algorithm [37] to simulate the SP-parse tree of a computation. Raman et al. [31] develop a race detection algorithm for async-finish parallelism and implement it in Habanero Java [7]. Raman et al. [32] developed an simple parallel algorithm that maintains the entire computations tree; while it provides good empirical results, it provides no theoretical guarantees. That implementation takes advantage of compiler optimizations to avoid instrumenting all memory access; we have not yet applied such optimizations. Lastly, in contrast to existing race detectors for fork-join computations, TARDIS [22, 24] does not explicitly keep track of the series-parallel relationships among strands. Instead, it employees a log-based access set and detects races by intersecting the access sets of logically parallel subcomputations at the join point. Moreover, it does not provide provably good time bounds as intersecting the access sets can lead to an asymptotic increase in the span of the computation.

Our runtime modification — the idea of switching between two deques — also exists in the BATCHER runtime [1]. BATCHER is a runtime scheduler augmented specifically for programs that use shared data structures. In particular, BATCHER allows the programmer to replace concurrent data structures by certain (non-concurrent) batched data structures.[7] The runtime automatically batches operations and allows workers that are blocked on a batch to help it complete. One possible way to parallelize SP-Order would apply BATCHER directly with a batched OM data structure. Unfortunately, roughly speaking, BATCHER itself adds $\Omega(\lg P)$ work/time to each data-structure access — therefore, even with an efficient OM data structure, it seems impossible to achieve an optimal SP-maintenance algorithm using BATCHER directly.

[7]A batched data structure supports, e.g., inserting a set of elements instead instead of inserting elements one by one.

7. CONCLUDING REMARKS

WSP-Order maintains series-parallel relationships in parallel and without any asymptotic slowdown for fork-join programs. In addition, our implementation of C-RACER provides good speedup in practice. One aspect we have not discussed is deletion from the order-maintenance data structures. In WSP-Order and in C-RACER, once a strand is inserted it will never be removed, even though it may never be used again. Performing some kind of garbage collection on such nodes may improve memory usage. In addition, WSP-Order is designed for series-parallel programs; it would be interesting to consider race detection for broader classes of parallelism, such as pipeline parallelism [23].

Acknowledgments

This research was supported in part by National Science Foundation grants CCF-1527692, CCF-1218017, CCF-1150036, CCF-1218188, and CCF-1314633. We thank the referees for their excellent comments.

8. REFERENCES

[1] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. SPAA '14, pages 84–95, New York, NY, USA, 2014. ACM.

[2] Todd R. Allen and David A. Padua. Debugging fortran on a shared memory machine. ICPP '87, pages 721–727, aug 1987.

[3] Arne Andersson. Improving partial rebuilding by using simple balance criteria. WADS '89, pages 393–402, London, UK, UK, 1989. Springer-Verlag.

[4] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. ESA '02, pages 152–164, London, UK, UK, 2002. Springer-Verlag.

[5] 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. SPAA '04, pages 133–144, New York, NY, USA, 2004. ACM.

[6] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46, 1999.

[7] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[8] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, October 1991.

[9] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. STOC '87, pages 365–372, New York, NY, USA, 1987. ACM.

[10] Paul F. Dietz. Maintaining order in a linked list. STOC '82, pages 122–127, New York, NY, USA, 1982. ACM.

[11] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware java runtime. PLDI '07, pages 245–255, New York, NY, USA, 2007. ACM.

[12] Perry A. Emrath and David A. Padua. Automatic detection of nondeterminacy in parallel programs. PADD '88, pages 89–99, New York, NY, USA, 1988. ACM.

[13] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[14] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. SPAA '97, pages 1–11, New York, NY, USA, 1997. ACM.

[15] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[16] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, December 1993.

[17] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[18] Igal Galperin and Ronald L. Rivest. Scapegoat trees. SODA '93, pages 165–174, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[19] Intel Corporation. CilkPlus/LLVM. http://cilkplus.github.io/, 2013.

[20] Intel Corporation. Intel® Cilk™ Plus Language Extension Specification, Version 1.2, September 2013. Document 324396-003US.

[21] Intel Corporation. An introduction to the cilk screen race detector., November 2013.

[22] Weixing Ji, Li Lu, and Michael L. Scott. Tardis: Task-level access race detection by intersecting sets. WoDet '13, Houston, TX, USA, 2013.

[23] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly pipeline parallelism. SPAA '13, pages 140–151, New York, NY, USA, 2013. ACM.

[24] Li Lu, Weixing Ji, and Michael L. Scott. Dynamic enforcement of determinism in a parallel scripting language. PLDI '14, pages 519–529, Edinburgh, United Kingdom, 2014. ACM.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[26] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. Supercomputing '91, pages 24–33, New York, NY, USA, 1991. ACM.

[27] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. PADD '93, pages 129–139, New York, NY, USA, 1993. ACM.

[28] Robert Netzer and Barton P. Miller. Detecting data races in parallel program executions. In *In Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, pages 109–129. MIT Press, 1989.

[29] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[30] Eli Pozniansky and Assaf Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, March 2007.

[31] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. RV'10, pages 368–383, Berlin, Heidelberg, 2010. Springer-Verlag.

[32] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. PLDI '12, pages 531–542, New York, NY, USA, 2012. ACM.

[33] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. SOSP '97, pages 27–37, New York, NY, USA, 1997. ACM.

[34] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. The cilkprof scalability profiler. SPAA '15, pages 89–100, Portland, Oregon, USA, June 2015. ACM.

[35] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[36] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. PPoPP '13, pages 299–300, New York, NY, USA, 2013. ACM.

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

[38] Intel Cilk Plus Development Team. Intrinsics for Low Overhead Tool Annotations. Technical report, November 2011.