Efficient Parallel Determinacy Race Detection for Two-Dimensional Dags

Yifan Xu
Washington University in St. Louis
xuyifan@wustl.edu

I-Ting Angelina Lee
Washington University in St. Louis
angelee@wustl.edu

Kunal Agrawal
Washington University in St. Louis
kunal@wustl.edu

Abstract

A program is said to have a determinacy race if logically parallel parts of a program access the same memory location and one of the accesses is a write. These races are generally bugs in the program since they lead to non-deterministic program behavior — different schedules of the program can lead to different results. Most prior work on detecting these races focuses on a subclass of programs with fork-join parallelism.

This paper presents a race-detection algorithm, 2D-Order, for detecting races in a more general class of programs, namely programs whose dependence structure can be represented as planar dags embedded in 2D grids. Such dependence structures arise from programs that use pipelined parallelism or dynamic programming recurrences. Given a computation with T_1 work and T_∞ span, 2D-Order executes it while also detecting races in $O(T_1/P + T_\infty)$ time on P processors, which is asymptotically optimal.

We also implemented PRacer, a race-detection algorithm based on 2D-Order for Cilk-P, which is a language for expressing pipeline parallelism. Empirical results demonstrate that PRacer incurs reasonable overhead and exhibits scalability similar to the baseline (executions without race detection) when running on multiple cores.

ACM Reference format:
Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal. 2018. Efficient Parallel Determinacy Race Detection for Two-Dimensional Dags. In *Proceedings of PPoPP '18: Principles and Practice of Parallel Programming, Vienna, Austria, February 24–28, 2018 (PPoPP '18),* 13 pages.
https://doi.org/10.1145/3178487.3178515

1 Introduction

A ***determinacy race*** [19] occurs when two or more logically parallel instructions access the same memory location, and at least one of them is a write.[1] A determinacy race are often bugs since they can lead to nondeterministic program behaviors. In this paper, we focus on performing race detection ***on the fly*** as the program executes; for a given program and input, we want to report a race if and only if the program contains a race for that input (regardless of schedule).

One can model the execution of a program as a ***directed acyclic graph*** (or ***dag*** for short), where a node represents a ***strand***, or a sequence of instructions without any parallel control, and an edge represents a control dependence. Given a currently executing strand u that reads a memory location ℓ, a race detection algorithm reports a race if there is a previously executed strand v such that v wrote to ℓ and v is logically in parallel with u (there is no path from v to u).

Many on-the-fly race detection algorithms [6, 20, 35, 40, 47, 48, 60] have been developed for ***fork-join*** programs — a form of parallelism supported by many concurrency platforms. It is exemplified by the `spawn` and `sync` constructs in Cilk dialects [12, 22, 25, 27, 31], the `async` and `finish` constructs in X10 [9] and Habanero dialects [4, 8], and the `task` pragma in OpenMP [42]. Fork-join programs can be represented as ***series-parallel dags*** [61] — these are dags with special structural properties that race-detection algorithms exploit to do efficient race-detection. In particular, for such dags, Utterback et al. [60] recently proposed a parallel algorithm for on-the-fly race detection that has an asymptotically optimal parallel running time.

Another popular structured dag model is ***two-dimensional dags*** (or ***2D dags***) — planar dags that can be embedded within a two-dimensional grid space.[2] Programs that generate 2D dags can arise from commonly used paradigms such as dynamic programming recurrences and linear pipelines. Linear pipelines, in particular, are a well-known parallel pattern used to parallelize applications and is widely supported [1, 11, 23, 24, 32, 33, 43, 43, 44, 46, 49–53, 56, 59], including notably Intel's Threading Building Blocks (TBB) [34], the `ordered` directive in OpenMP [42], and Cilk-P [28, 29], an extension to Cilk designed specifically for linear pipelines. It has been shown that a program with line pipelines can be scheduled efficiently using work stealing [28, 29].

PPoPP '18, February 24–28, 2018, Vienna, Austria
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178515

[1] In contrast, a ***data race*** occurs when the atomicity of critical sections is violated. A determinacy race is sometimes referred to as a ***general race*** [39]. This paper focuses on detecting determinacy races, and henceforth when we say race, we mean determinacy race.

[2] We formally define 2D dags that our algorithm targets in Section 2.

To our knowledge, Dimitrov et al. [14] provide the known algorithm for on-the-fly race detection for 2D dags. This algorithm must execute the program sequentially and it has never been implemented and evaluated in practice. This paper provides an efficient, parallel, and practical algorithm to do race detection on 2D-dags. In particular, we make the following contributions:

2D-Order (Section 2): We present ***2D-Order***, a provably correct and efficient race detection algorithm for 2D dags, that has an asymptotically optimal parallel running time. Given a program with ***work*** T_1 — the amount of time it takes to run on one processor — and ***span*** T_∞ — the amount of time it takes to run on infinitely-many processors, 2D-Order can detect races while executing the computation on P processors with expected time $O(T_1/P + T_\infty)$. This bound is asymptotically optimal, since this is the best one can do when executing the same program without race detection. 2D-Order provides a strong correctness guarantee — it reports a race if and only if the program has a race on that input.

Like prior work, 2D-Order has two main components. (1) ***SP-maintenance*** component maintains enough information to answer the question: Is strand u is logically in parallel with a strand v. As it executes the computation, 2D-Order maintains two total orders of the strands it has encountered. In Section 2, we define these two specific orders, show how to maintain them on the fly, and prove that maintaining these two orders suffices to answer this question. (2) ***Memory access history*** component keeps track of (a subset of) previous reader and writer strands for each memory location ℓ. For parallel race-detection in dags with no structural properties, one must keep one writer and an unbounded number readers. For series-parallel programs, Mellor-Crummey [35] proved that two readers and one writer suffice. We extend this result and show that two readers and one writer suffice for 2D-dags as well, which directly follows from the fact that SP-relationships can be maintained with two orders.

In order to maintain the two orders, we can use two order-maintenance (OM) data structures [5, 13]. We adapt the technique described in Utterback [60] which describes an OM data structure plus scheduler combination that supports concurrent accesses into the data structure while providing optimal running time.

Generalizing 2D-Order (Section 3): The algorithm presented in Section 2 assumes that when a strand u (a node in the dag) is executed, we know how many children u has. This assumption may not hold for platforms that generate the pipelined dag dynamically. In Section 3, we present a variant of 2D-Order which only assumes that a node v knows its parents when it executes; this assumption is generally true since v can only execute after all its parents have executed. This variant has the same performance bound.

PRacer: Implementing and Evaluating 2D-Order in Cilk-P (Sections 4 and 5): The algorithms described in Sections 2 and 3 are formulated in terms of traversing a 2D dag as the computation unfolds. In Section 4, we show how one can apply 2D-Order traversal to the language constructs provided by Cilk-P [28, 29]. Cilk-P is an extension to the Cilk language that supports linear pipelines with a provably efficient work-stealing scheduler. Cilk-P is an interesting case study. Unlike most other systems, Cilk-P supports "on-the-fly" pipeline parallelism, which allows the programmer to dictate how the 2D dag may be embedded in the two-dimensional grid. Due to this, Cilk-P has a particular quirk that when a node executes, it does not automatically know its parents' identities. Since it is essential to identify parents, 2D-Order performs additional bookkeeping to enable this. Consequently, when applied to Cilk-P's computation dag on P processors, 2D-Order runs in expected time $O(T_1/P + \lg k \cdot T_\infty)$, where k is the vertical length of the two-dimensional grid. In practice, this additional $\lg k$ overhead is typically small — in the evaluated benchmarks, k ranges from 3 to 71.

This additional overhead is only due to the particular quirks of Cilk-P's language constructs and would not apply for systems such as Intel TBB, where an executed strand can easily identify its parents.

We have implemented PRacer, a prototype implementation of 2D-Order race detection algorithm applied to Cilk-P. Section 5 empirically evaluates the overhead of PRacer and shows that it incurs virtually no overhead for SP-maintenance and achieves similar scalability compared to applications' baseline executions without race detection.

2 2D-Order Algorithm

We now describe the basic 2D-Order algorithm. In this section, we make two simplifying assumptions: (1) We assume that a node u's children are known as soon as u finishes executing; and (2) There are no redundant edges — and edge from (u, v) is removed if there is already (a different) directed path from u to v. We will remove these assumptions in the next sections. We first provide some basic notation before describing the 2D-Order's algorithm for series-parallel maintenance and proving its correctness. Then, we describe what information is kept in the access history and how 2D-Order checks for races. Finally, we prove the performance bound.

Notation and Definitions

We say $x \prec y$ iff there is a (non-empty) path from x to y in the dag. $x \preceq y$ iff either $x \prec y$ or $x = y$. We say $x \parallel y$ iff there is no path from x to y or from y to x.

Definition 2.1. A ***2D dag*** is a planar directed acyclic graph with the following properties:

1. It has a unique source node s with no incoming edges and a unique sink node t with no outgoing edges.
2. Each node has at most two incoming and at most two outgoing edges. Edges are labeled as pointing either rightwards or downwards.

This definition implies that each node can have at most two children — the down-child of a node v is denoted by

$v.dchild$ and the right-child is denoted by $v.rchild$. Similarly, the up parent of v is denoted by $v.uparent$ and the left parent is denoted by $v.lparent$.

Definition 2.2. Given two distinct nodes x and y, a node v is their ***common ancestor*** if $v \preceq x$ and $v \preceq y$. A node z is their ***least common ancestor***, denoted by $lca(x, y)$, if for all common ancestors v of x and y, we have $v \preceq z$.

By definition of a 2D dag, a unique least common ancestor exists for any two nodes (proven in Lemma 2.9). The following lemma states that if $x \parallel y$, then their lca has two children, and x follows from one while y follows from the other.

Lemma 2.3. *For two nodes x and y, say $x \parallel y$ and $z = lca(x, y)$. Then we have (1) z has two children; (2) if $z.dchild \preceq x$ then $z.dchild \parallel y$, $z.rchild \preceq y$, and $z.rchild \parallel x$.*

Proof. Suppose that, for contradiction, $w \preceq x$ and $w \preceq y$ where w is a child of z; by Definition 2.2 $z \neq lca(x, y)$. □

This lemma allows us to relate any two parallel nodes.

Definition 2.4. Given two nodes x and y where $x \parallel y$. Let $z = lca(x, y)$. Then, x is ***down of*** ($\parallel_D$) y iff $z.dchild \preceq x$ & $z.rchild \preceq y$, and x is ***right of*** y ($\parallel_R$) iff $z.dchild \preceq y$ & $z.rchild \preceq x$.

We now make some straightforward structural observations. (1) For distinct nodes x and y, exactly one of the following four conditions hold: $x \prec y$, $y \prec x$, $x \parallel_D y$ or $y \parallel_D x$. (2) Given a node x with two children, $x.dchild \parallel_D x.rchild$.

2.1 SP-Maintenance in 2D-Order Algorithm

```
Function Insert-Down-First($v$)
    if $v.rchild$ exists then
        if $v.rchild.uparent$ not exists then
            OM-Insert(OM-DownFirst, $v$, $v.rchild$);
        end
    end
    if $v.dchild$ exists then
        OM-Insert(OM-DownFirst, $v$, $v.dchild$);
    end
end
Function Insert-Right-First($v$)
    if $v.dchild$ exists then
        if $v.dchild.lparent$ not exists then
            OM-Insert(OM-RightFirst, $v$, $v.dchild$);
        end
    end
    if $v.rchild$ exists then
        OM-Insert(OM-RightFirst, $v$, $v.rchild$);
    end
end
```

Algorithm 1: 2D-Order

2D-Order maintains two total orders on all strands using order-maintenance data structures. An ***order-maintenance (OM)*** data structure D maintains a total order of elements and provides the following operations.

- OM-Precedes(D, x, y): Given pointers to x and y, return *true* iff x precedes y in the total order kept by D.
- OM-Insert(D, x, y): Given a pointer to an existing element x, splice-in a new element y *immediately* after x in the total order. Thus, x and all its predecessors of x are before y in the total order, while all successors of x are after y.

2D-Order keeps two OM data structures — called OM-RightFirst and OM-DownFirst — to maintain two different orders on all the nodes in the 2D dag. The OM data structures for both orders are initialized by inserting the source node s as the first node. Subsequently, it executes nodes of the dag in any valid serial or parallel order — that is, a node can be executed when it's predecessors have finished executing. After executing each node v, 2D-Order calls the two functions shown in Algorithm 1.

Function `Insert-Down-First(`v`)` inserts v's children into the OM-DownFirst data structure. Immediately after this function is executed, the following will be true in the OM-DownFirst order: (a) If v has a down child v_D, then v_D will be immediately after v. (2) If v has a right child v_R and v_R doesn't have an up parent, then v_R will be immediately after v_D (if v_D doesn't exist, then v_R will be immediately after v). The symmetric invariant is true for the function `Insert-Right-First(`v`)`.

In other words, for any node u, its up parent is "responsible" for inserting it into the OM-DownFirst data structure and its left parent is "responsible" for inserting it into the OM-RightFirst data structure. If u doesn't have one of the parents, however, then u's other parent takes over the corresponding responsibility and inserts u immediately after its other child (or after the parent itself if the other child doesn't exist). It should be clear that each node u is inserted into each OM data structure exactly once and these insertions happen before u itself is executed.

To simplify notation, we say that $x \rightarrow_D y$ if x occurs before y in the OM-DownFirst data structure (that is, if OM-Precedes(OM-DownFirst, x, y) returns true). Similarly, we say $x \rightarrow_R y$ if OM-Precedes(OM-RightFirst, x, y) returns true. Note that since the algorithm never swaps the order of the nodes once they are inserted, the answer returned will be consistent once both x and y are inserted.

2.2 OM-DownFirst and OM-RightFirst Maintain Series-Parallel Relationships

We will now prove that these two total orders are sufficient to fully specify the partial order of the dag. The following theorem, which we prove in the remaining subsection, shows that given any two nodes x and y, we can determine the relationship between them just by looking at the total orders maintained by OM-DownFirst and OM-RightFirst; if x is before y in both orders, then $x \prec y$; if y is before x in both orders, then $y \prec x$; otherwise $x \parallel y$.

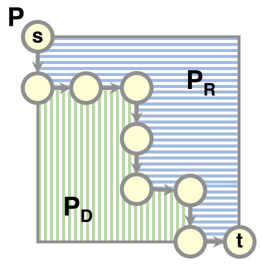

Figure 1. A path P divides a 2D dag into two regions, P_R (shaded with horizontal lines) and P_D (shaded in vertical lines).

Theorem 2.5. *Given nodes x and y in OM-DownFirst and OM-RightFirst, $x \prec y$ iff $x \rightarrow_D y$ and $x \rightarrow_R y$.*

We first prove some structural properties of 2D dags. Lemma 2.6 (stated without proof) says that any subdag G' of a 2D dag G that has a single source and a single sink is also a 2D dag. Then, we prove that any source to sink path cuts 2d dags into disjoint graphs with certain properties (Lemmas 2.7 and 2.8).

Lemma 2.6. *Consider two nodes a and b in a 2D dag G where $a \prec b$. Construct a subdag G' such that $x \in G'$ if $a \preceq x \preceq b$ and edge $(x, y) \in G'$ if $(x, y) \in G$. Then G' is a 2D dag with source a and sink b.*

Consider a path P from the source s of a 2D dag to its sink t. We use this path to divide all nodes of the dag into three subsets P, P_R and P_D. P contains all the nodes on the path. Note that for all $u \in P$, at least one of u's children (unless $u = t$) and one of u's parents (unless $u = s$) is also in P. Consider a node $u \notin P$ — since $s \in P$, u has at least one ancestor in P. Say q is the ancestor of u which is topologically latest in the path P. Since nodes on P are totally ordered, there is necessarily this latest node. Then, $u \in P_D$ if it follows from q's down child — that is, if $q.dchild \preceq u$. Similarly, $u \in P_R$ if $q.rchild \preceq u$. The intuitive meaning of P_R and P_D is shown in Figure 1 where path P cleanly divides the graph into two "contiguous regions." We now prove that the definition matches this intuitive meaning.

Lemma 2.7. *For any path P, $P_R \cap P_D = \emptyset$ and any path from node u to node v where $u \in P_R$ and $v \in P_D$ must include some node on P.*

Proof. First, we prove disjointness. Consider $u \notin P$ and say q is u's ancestor which is topologically latest in P; since at least one of q's children must be in P, u cannot follow from both its children. Now, assume for contradiction that there is a path from u to v that has no node in P. Then the latest ancestor of u and v on P must be the same node — which is a contradiction since $u \in P_R$ and $v \in P_D$. □

We omit the proof of Lemma 2.8 due to space constraints; however, the intuition should be clear from Figure 1:

Lemma 2.8. *For any node $u \in P$,*

1. *If u has two children and $u.rchild \in P$, then $u.dchild \in P_D$. (Similarly, if $u.dchild \in P$, then $u.rchild \in P_R$.)*
2. *If u has two parents, if $u.lparent \in P$, then $u.uparent \in P_R$. (Similarly, if $u.uparent \in P$, then $u.lparent \in P_D$.)*

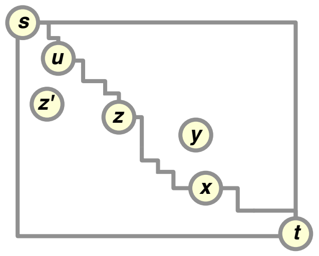

Figure 2. Figure for Lemma 2.9. Assume two lcas for x and y exist, namely z and z', and let u be a lca of z and z'.

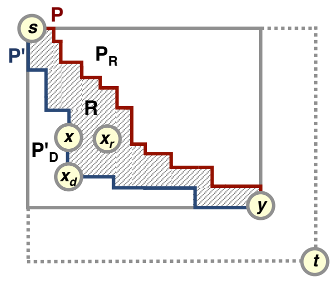

Figure 3. Figure for Lemma 2.10. Path P is shown in red, and path P' is in blue. The shaded region is R. The node x_d is $x.dchild$ and the node x_r is $x.rchild$.

The first statement is obvious from definition, because $u \in P$ and u must be the latest ancestor of $u.dchild$ in P. The easy way to see the second statement is to notice that if we flip the direction of all edges and rotate the dag, then source becomes sink, $u.uparent$ becomes $u.dchild$, and $u.lparent$ becomes $u.rchild$, and P_R becomes P_D. To formally prove it, one can induct on the nodes on the path.

We can now use Lemma 2.7 to prove that any two nodes have a unique lca.

Lemma 2.9. *Given two nodes x and y which are in parallel, $lca(x, y)$ exists uniquely.*

Proof. For the purpose of contradiction, assume two lcas exist, named z and z'. By the fact that they are both $lca(x, y)$, they must be in parallel with each other. Wlog,[3] assume $z' \parallel_D z$ and let $u = lca(z', z)$. Also wlog assume that $x \parallel_D y$ with respect to z (i.e., x follows from $z.dchild$). Construct a path P that goes from the source s to u to $u.rchild$ to z to $z.dchild$ to x to the sink t; such a path exists by the property of 2D dags and lcas. Then, as shown in Figure 2, $z' \in P_D$ (since it follows from $u.dchild$) and $y \in P_R$ (since it follows from $z.rchild$). However, since $z' = lca(x, y)$, there must be a path P' from z' to y, which must cross P by Lemma 2.7. If P' cross P before z, it contradicts with the fact that no path exists between z and z'. If P' cross P after z, it contradicts with the fact that $z = lca(x, y)$. Thus, the $lca(x, y)$ must be unique. □

We now use these paths to prove a lemma about the insertion order between nodes.

[3]Wlog stands for "without loss of generality."

Lemma 2.10. *At any point during the execution of 2D-Order, given node x which has two children. If $x.dchild \notin$ OM-DownFirst, then for any y such that $x.dchild \prec y$ and $x.rchild \parallel y$, $y \notin$ OM-DownFirst.*

Proof. Consider the 2D subdag G' with source s and sink y. Let S be the set of all the nodes on paths from s to y. Now we construct path P using nodes from set S: for any node u such that $u \in S$ and $u \notin P$, we have $u \in P_D$. Intuitively, P follows the "top-most right-most" path among all paths from s to y.

First we prove $x \in P$. Suppose $x \notin P$, then by the constraint of path P, we have $x \in P_D$. Since $x.rchild \notin P$ because $x.rchild \parallel y$, we also have $x.rchild \in P_D$. Now construct a path P' from s to x to $x.dchild$ to y. By Lemma 2.8, $x.rchild \in P'_R$. Let $R = P'_R \bigcap P_D$ (the shaded region in Figure 3). Now consider the whole 2D dag G with source s and sink t, which includes R. Since $x.rchild \in R$ (the shaded region) and $t \notin R$, a path from $x.rchild$ to t must cross with either path P P'. Since both P and P' ends at y, this means $x.rchild \prec y$, which contradicts with the original assumption that $x.rchild \parallel y$. Therefore, we are guaranteed that $x \in P$.

Since P is a continuous path and $x.rchild \notin P$, we must have $x.dchild \in P$. Now we show P is actually an insertion chain, which means for any node w in P except for s, w is inserted by its parent in P.

Let v and w be two consecutive nodes in P and $v \prec w$. For the purpose of contradiction, let's assume that w is not inserted by v. According to the Down-First part of algorithm 1, this can only happen when w has two parents in G and $v = w.lparent$. Since $v \in P$, we have $v.uparent \in P_R$ by Lemma 2.8. However, since $v.uparent \in S$, it is guaranteed that either $v.uparent \in P$ or $v.uparent \in P_D$, which leads to a contradiction. Thus, P is an insertion chain. Since we know that $x.dchild \in P, y \in P$, and $x.dchild \prec y$, y cannot be inserted into OM-DownFirst before $x.dchild$. □

We can now prove the two important properties of down-first order — namely that $x \rightarrow_D y$ if $x \parallel_D y$ or if $x \prec y$.

Lemma 2.11. *At any point during the execution of 2D-Order, given nodes x and y in OM-DownFirst. If $x \parallel_D y$, then $x \rightarrow_D y$.*

Proof. We prove this lemma by induction. Suppose lemma is true before insertion is invoked on a node y. Consider any x in OM-DownFirst. We will show if $x \parallel_D y$, then $x \rightarrow_D y$. Let $z = lca(x, y)$; we have $z.dchild \preceq x$ and $z.rchild \preceq y$ (The case where $y \parallel_D x$ is similar.)

1. y has one single parent w: We first consider the case where $z = w$ and $z.rchild = y$. Since $z.dchild$ has not been inserted in OM-DownFirst yet, and since x must follow from $z.dchild$, from Lemma 2.10, we know x has not been inserted. So this case is trivial. Say $z.rchild \neq y$; then, we have $z = lca(w, x)$ and $x \parallel_D w$. By inductive hypothesis, we know $x \rightarrow_D w$. Because y is inserted immediately after w, therefore it's guaranteed that $x \rightarrow_D y$.

2. y has two parents: According to algorithm 1, y is inserted immediately after its up parent $y.uparent$, say w. We now argue that $x \parallel_D w$. Let $z' = lca(w, x)$. We know that $z' \preceq z$. If $z' = z$, then we are done. For the rest of the proof, we assume that $z \parallel w$; therefore, $z' = lca(z, w)$ since z' is an ancestor of z.

Assume for contradiction that $w \parallel_D x$; therefore $z'.dchild \preceq w$. Therefore, $z'.rchild \preceq z$. Consider the 2D-dag G' with source z' and sink t and consider a path P that goes from z' to $z'.dchild$ to w to y to t. We know that $z'.rchild \in P_R$ (from Lemma 2.8); therefore, $z \in P_R$ since otherwise, the path from z' to z will cross P at some node a and this node $a = lca(z, w)$ instead of z'. However, we also know that $y.lparent \in P_D$ (from Lemma 2.8) and $z \preceq y.lparent$. Therefore, the path from z to $y.lparent$ must cross P at some node b and this would mean that z is an ancestor of w which contradicts the assumption.

Therefore $x \parallel_D w$ and by IH, $x \rightarrow_D w$; therefore, when y is inserted immediately after w, we have $x \rightarrow_D y$. □

Lemma 2.12. *At any point during the execution of 2D-Order, given nodes x and y in OM-DownFirst. If $x \prec y$, then $x \rightarrow_D y$.*

Proof. We prove this lemma by induction. Suppose lemma is true before insertion is invoked on a node y. Consider any node x in OM-DownFirst.

We first show if $y \prec x$ then $y \rightarrow_D x$. Let w be the parent that inserted y. Since $y \prec x$, $w \prec x$. By IH, $w \rightarrow_D x$ before the insertion, and thus $w \rightarrow_D y \rightarrow_D x$ after the insertion.

Now we show if $x \prec y$, then $x \rightarrow_D y$.

1. y has one single parent w: In this case, y is inserted by this parent w immediately after w. Therefore, clearly, $w \rightarrow_D y$. Also, if $x \prec y$, then $x \preceq w$ since w is y's only parent. If $x = w$, we are done. If not, according to the inductive hypothesis, we have $x \rightarrow_D w$. Therefore, we know $x \rightarrow_D y$.

2. y has two parents: We know y is inserted immediately after $y.uparent$. If $x \preceq y.uparent$, we can apply the same argument as in the single parent case. Now consider the case that $x \parallel y.uparent$ and $x \preceq y.lparent$. We will show that $x \parallel_D y.uparent$ and by Lemma 2.11, we then have $x \rightarrow_D y.uparent$, which leads to $x \rightarrow_D y$.

For the purpose of contradiction, let's assume that $y.uparent \parallel_D x$ and let $z = lca(x, y.uparent)$. Consider the subdag with source z and sink t and construct a path P from z to $z.rchild$ to x to y to t. Then by Lemma 2.8, we have $y.uparent \in P_R$ and $z.dchild \in P_D$. Since $y.uparent \parallel_D x$, there must be a path P' from $z.dchild$ to $y.uparent$, which must cross P by Lemma 2.7, which contradicts the assumption that $z = lca(x, y.uparent)$. Thus we must have $x \parallel_D y.uparent$ and thus $x \rightarrow_D y$. □

Symmetrically, one can prove the following lemmas:

Lemma 2.13. *At any point during the execution of 2D-Order, given nodes x and y in OM-RightFirst. If $x \prec y$, then $x \rightarrow_R y$.*

Lemma 2.14. *At any point during the execution of 2D-Order, given nodes x and y in OM-RightFirst. If $x \parallel_R y$, then $x \rightarrow_R y$.*

Now we can prove the main result:

Proof of Theorem 2.5. From Lemma 2.12 and Lemma 2.13, it's straightforward to see that if $x \prec y$, then $x \rightarrow_D y$ and $x \rightarrow_R y$. For the other direction, suppose $x \parallel y$ when $x \rightarrow_D y$ and $x \rightarrow_R y$. Wlog, say $y \parallel_D x$. Then we have $y \rightarrow_D x$ by Lemma 2.11, which contradicts to $x \rightarrow_D y$. □

2.3 Checking races and updating access history

```
// Called when a strand $r$ read memory location $l$
Function Read($r, l$)
    if Precedes ($lwriter(l), r$) is false then
        ReportRace ( );
    end
    if OM-PRECEDES(OM-RightFirst, $dreader(l), r$) then
        $dreader(l) = r$;
    end
    if OM-PRECEDES(OM-DownFirst, $rreader(l), r$) then
        $rreader(l) = r$;
    end
end
// Called when a strand $w$ wrote to memory location $MemLoc$
Function Write($w, l$)
    if Precedes ($lwriter(l).w$) is false
    or Precedes ($dreader(l), w$) is false
    or Precedes ($rreader(l), w$) is false then
        ReportRace ( );
    end
    $lwriter(l) = w$;
end
// When called, we have either $u \prec v$ or $u \parallel v$, but never $v \prec u$
Function Precedes($u, v$)
    if OM-PRECEDES(OM-DownFirst, $u, v$)
    and OM-PRECEDES(OM-RightFirst, $u, v$) then
        return true
    end
    return false
end
```

Algorithm 2: Access history

We now describe how we do race detection using this algorithm — the code is shown in Algorithm 2. For each memory location ℓ, our algorithm stores at most one previous writer node — called ***last writer*** $lwriter(\ell)$ — this is simply the last node that wrote to this memory location. If a set of reader nodes R_ℓ have read this memory, the algorithm stores up to two reader nodes: (1) ***downmost reader*** $dreader(\ell)$: For all $r \in R_\ell$, either $r \preceq dreader(\ell)$ or $r \parallel_R dreader(\ell)$; and (2) ***rightmost reader*** $rreader(\ell)$: For all $r \in R_\ell$, either $r \prec rreader(\ell)$ or $r \parallel_D rreader(\ell)$.

When a node u tries to write location ℓ, it uses the OM-DownFirst and OM-RightFirst data structures to check whether either $dreader(\ell) \parallel u$, $rreader(\ell) \parallel u$ or $lwriter(\ell) \parallel u$. If so, we report a race. In either case, u is now last writer $lwriter(\ell)$. When node u tries to read location ℓ, it uses the OM-DownFirst and OM-RightFirst data structures to check whether $lwriter(\ell) \parallel u$. If so, we report a race. In either case, u now checks if $dreader(\ell) \rightarrow_R u$; if so, u is the new downmost reader $dreader(\ell)$. Similarly, if $rreader(\ell) \rightarrow_D u$, then u is the new $rreader(\ell)$.

Theorem 2.15. *2D-Order never reports false races and for racy programs, reports at least one race.*

This is the standard correctness guarantee for on-the-fly race detection algorithms. The proof mostly follows from previous results — with the exception of one wrinkle. It is always sufficient to store a single writer in access history; however, it is not always sufficient to store two readers. In particular, for general dags, one has to store all parallel reads that happened since the last write. Mellor-Crummey [35] proved that for series-parallel dags, it is sufficient to record two readers. We will now show that for 2D dags, it is also sufficient to store two readers — in particular, the downmost and the rightmost readers.

First, let us notice that at any point in the execution $rreader(\ell)$ (and $dreader(\ell)$) is unique (or null if no node has read ℓ yet). To see this, note that from Algorithm 2, line 9 and line 10, $rreader(\ell)$ is simply the last node in the order maintained by OM-DownFirst that read ℓ.

Theorem 2.16. *At any point during the execution of a 2D dag, let R_ℓ be the set of nodes that have read memory location ℓ and w be any other node. We have $r \prec w$ for all $r \in R_\ell$ if and only if $dreader(\ell) \prec w$ & $rreader(\ell) \prec w$.*

Proof. It is clear that if all $r \in R_\ell$ precede w then so do $dreader(\ell)$ and $rreader(\ell)$ since they belong to the set R_ℓ.

Now say that $dreader(\ell) \prec w$ & $rreader(\ell) \prec w$. For any $r \in R_\ell$, we have, $r \rightarrow_R dreader(\ell) \rightarrow_R w$ (the first arrow is by definition of downmost reader and the second from Theorem 2.5). Similarly, we have $r \rightarrow_D rreader(\ell) \rightarrow_D w$. Therefore r is before w in both OM-DownFirst and OM-RightFirst orders; by Theorem 2.5, we know that $r \prec w$. □

2.4 Performance of 2D-Order

Theorem 2.17. *For a 2D dag G with work T_1 and span T_∞, we can run do race detection using 2D-Order in time $T_1/P + T_\infty$ time on P processors.*

First, each node is inserted at most once in OM data structures and every memory access requires a constant number queries to OM data structures. If inserts and queries to OM data structures took $O(1)$ time, then the work of the program augmented with 2D-Order is $O(T_1)$ and the span is $O(T_\infty)$.

Sequentially, an OM-data structure can be implemented for $O(1)$ cost per operation (amortized) [5, 13]. This immediately gives us an $O(T_1)$ time (optimal) sequential algorithm. This slightly improves the best previous result [14], which has a multiplicative overhead of the inverse Ackermann's function (which is, admittedly, small in practice).

To get parallel performance, we need an OM implementation that supports concurrent operations. No general $O(1)$-time-per-operation concurrent OM data structure is known. Utterback et al. [60] provide an algorithm (containing modified OM data structure and work-stealing scheduler) for programs that access OM data structures in a ***conflict-free*** way. In particular, if a parallel program guarantees that two logically parallel strands will never try to insert immediately after the same node, then they show the following result (it is not explicitly stated, but is implied from their proofs).

Lemma 2.18. ***From [60]*** *A parallel program with work T_1 and span T_∞ which accesses into OM data structure(s) in a conflict-free way can be executed in time $O(T_1/P + T_\infty)$ time.*

Note that 2D-Order follows the conflict-free restriction since all inserts after node v occur when v executes. Therefore, Theorem 2.17 follows directly from this lemma. Note that this performance bound only holds if we use the particular implementations of both the work-stealing scheduler and the OM data structure described in Utterback [60]. In Section 5, we briefly describe how we adapted this scheduler for Cilk-P runtime system.

3 Generalizing 2D-Order

In Section 2, we made two assumptions: (1) When we execute a node, we already know both its children and whether these children have their other parent. In practice, we may not have this information until we encounter the child node. (2) There are no redundant edges.

Algorithm 3 shows a variant of 2D-Order — these functions are called immediately before executing node v. Here, when v is executed, instead of inserting its real children, 2D-Order creates two placeholder nodes for both of v's children (denoted as $dchild_h$ and $dchild_h$). It will insert both nodes into OM-DownFirst and OM-RightFirst orders. As seen on lines 7, 8, 16, and 17 the order after all insertions is $v \rightarrow_D v.dchild_h \rightarrow_D v.rchild_h$ and $v \rightarrow_R v.rchild_h \rightarrow_R v.dchild_h$. This is consistent with Algorithm 1 except here we assume that both children exist and always insert them regardless of the presence of the other parent.

When a node is executed, it finds its corresponding placeholder nodes by accessing its parents. If v has only one parent, it has only one placeholder node in each data structure, and 2D-Order simply use this placeholder node to represent v in the future. Now consider a node v that has two parents. Both parents will insert a placeholder node to represent v in OM-DownFirst and OM-RightFirst data structures, possibly at different positions. When v is executed, 2D-Order chooses one of these dummies as the "real" one (and different ones in OM-DownFirst and OM-RightFirst) which will be used henceforth to represent v when accessing each order. In particular, whenever we access OM-DownFirst, the placeholder inserted by v's up parent will represent v.[4] Correspondingly, when we access OM-RightFirst, the placeholder inserted by v's left parent will represent v. The access history and queries are not affected.

```
1  Function Insert-Down-First($v$)
2      if $v.uparent$ exists then
3          $dCurr = v.uparent.dchild_h$;
4      else
5          $dCurr = v.lparent.rchild_h$;
6      end
7      OM-Insert(OM-DownFirst, $dCurr$, $v.rchild_h$);
8      OM-Insert(OM-DownFirst, $dCurr$, $v.dchild_h$);
9  end
10 Function Insert-Right-First($v$)
11     if $v.lparent$ exists then
12         $rCurr = v.lparent.rchild_h$;
13     else
14         $rCurr = v.uparent.dchild_h$;
15     end
16     OM-Insert(OM-RightFirst, $rCurr$, $v.dchild_h$);
17     OM-Insert(OM-RightFirst, $rCurr$, $v.rchild_h$);
18 end
```

Algorithm 3: variant 2D-Order

The code for handling redundant edges is not shown, but is straightforward. When a node x has two parents, it first checks if either of them precede the other (using OM-DownFirst and OM-RightFirst) and if so, it ignores the redundant edge.

Lemma 3.1. *Algorithm 3 has the same correctness and performance properties as Algorithm 1.*

The intuition behind the omitted proof is that Algorithm 3 maintains the same order as Algorithm 1 — node v finds its correct representatives right before it is executed. Before v executes, the placeholder nodes for v will never be used in any queries or to insert any other nodes. For the performance guarantee, notice that each function call does at most twice as many inserts as Algorithm 1.

4 PRacer: Race Detection for Cilk-P

This section describes PRacer, the particular implementation of 2D-Order when applied to Cilk-P [28, 29]. Race-detection for Cilk-P is an interesting case study because Cilk-P's language constructs allow for much more dynamism in the structure of the pipeline. Due to the particular quirks to Cilk-P's pipelines, PRacer incurs an additional $\lg k$ overhead on the span term, where k is the vertical length of the 2D dag. This section reviews Cilk-P's support for pipeline parallelism, presents PRacer in terms of Cilk-P's pipeline constructs, and explains the additional performance overhead.

[4]The placeholder inserted by the v's left parent will never be accessed in OM-DownFirst if v also has an up parent — this node becomes a dummy. As an optimization, we can remove this node from OM-DownFirst; however, this has no bearing on the theoretical correctness or performance.

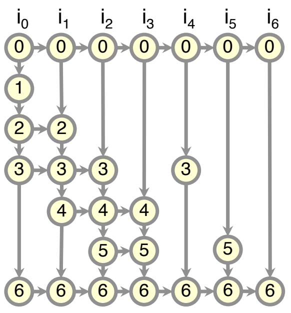

Figure 4. An example of the kind of 2D dag Cilk-P can generate. A node presents a strand, and an edge denotes dependence between two strands. The iteration numbers are denoted above, and the numbers in the nodes denote the stage numbers.

4.1 Cilk-P's Support for Pipeline Parallelism

From a programmer's perspective, a linear pipeline is simply a loop over a stream of input elements, where each loop ***iteration*** i processes the i^{th} element of the input stream. The loop body encodes the sequence of ***stages*** — abstract functions through which input elements are processed. These pipelines allow for parallel execution since the execution of iterations can overlap in time

Cilk-P extends Cilk with three keywords: `pipe_while`, `pipe_stage`, and `pipe_stage_wait`. The keyword `pipe_while` denotes a loop that can be executed in parallel in a pipelined fashion. The first stage of each iteration is stage 0 and `pipe_while` ensures that there are sequential dependences across stage 0 of all iterations; that is, stage 0 of iteration i does not begin until stage 0 of iteration i-1 completes. The keywords `pipe_stage` and `pipe_stage_wait` are used inside the body of a `pipe_while` loop to denote stage boundaries. By default, the execution of these constructs in stage s of iteration i ends stage s and advances to stage s+1 of iteration i. Keyword `pipe_stage_wait` is used to enforce dependences between adjacent iterations. Ending a stage s of iteration i with `pipe_stage_wait` enforces that execution of stage s+1 of iteration i does not begin until stage s+1 of iteration i-1 finishes. Finally, `pipe_while` implicitly has a cleanup stage at the end of every iteration that occurs sequentially across iterations.

The structure of the pipeline for Cilk-P programs is determined dynamically at runtime. The `pipe_stage` and `pipe_stage_wait` statements can be enclosed within other control constructs, which allows programmers to dynamically vary the number of stages and enforce dependences based on the input to the iteration. Furthermore, `pipe_stage` and `pipe_stage_wait` optionally take a ***stage number***, an integer argument to name the stage that the statement is advancing to. This gives the programmer the flexibility to dynamically determine the label of stages and skip stages.

Cilk-P constructs can generate the example dag shown in Figure 4. Each iteration is a vertical line. There are sequential dependences across the first and last stages of all iterations, as dictated by `pipe_while`. The stages of an iteration form a chain, and horizontal dependences are enforced by `pipe_stage_wait`. By naming stages in a certain way, the programmer can skip stages or dictate that the stages to be labeled certain way. Still, Cilk-P's pipeline constructs always generate a dag that satisfies Definition 2.1 in Section 2.

Cilk-P incorporates a work-stealing scheduler that can schedule the resulting computation efficiently. Given a computation with T_1 work and T_∞ span, Cilk-P schedules the computation on P processors in expected time $T_1/P+O(T_\infty)$.[5]

4.2 PRacer: Applying 2D-Order to Cilk-P

```
Function StageFirst($i$)
    if $i$ is 0 then
        $dCurr = rCurr = source$;
    else
        $dCurr = rCurr = stage[i-1][0].rchild_h$;
    end
    InsertPlaceHolder ($dCurr$, $rCurr$, $stage[i][0]$);
end
Function StageNext($i, s$)
    $dCurr = rCurr = stage[i][s-1].dchild_h$;
    InsertPlaceHolder ($dCurr$, $rCurr$, $stage[i][s]$);
end
Function StageWait($i, s$)
    $dCurr = stage[i][s-1].dchild_h$;
    $left$ = FindLeftParent ($i, s$);
    if $left \neq -1$ then
        $rCurr = stage[i-1][left].rchild_h$;
    else
        $rCurr = stage[i][s-1].dchild_h$;
    end
    InsertPlaceHolder ($dCurr$, $rCurr$, $stage[i][s]$);
end
Function InsertPlaceHolder($dCurr, rCurr, stage$)
    OM-Insert(OM-DownFirst, $dCurr$, $stage.rchild_h$);
    OM-Insert(OM-DownFirst, $dCurr$, $stage.dchild_h$);
    OM-Insert(OM-RightFirst, $rCurr$, $stage.dchild_h$);
    OM-Insert(OM-RightFirst, $rCurr$, $stage.rchild_h$);
end
```

Algorithm 4: 2D-Order for Cilk-P

Algorithm 4 shows the pseudocode for applying 2D-Order to Cilk-P's pipeline constructs. In Cilk-P, nodes do not know if they have a right child when they execute. Stage s of iteration i does not know if stage s of iteration i+1 will depend on it; we only find out that this dependence exists when (and if) stage s-1 of iteration i+1 calls `pipe_stage_wait`. Therefore, like in Algorithm 3, we must employ placeholder nodes. The function `StageFirst` is called before executing stage 0 of an iteration and is similar to Algorithm 3. The main difference is that, since stage 0 has no *uparent*, it knows to use the $rchild_h$ from its *lparent* (stage 0 of the previous iteration) as its representative. The function `StageNext` is called when `pipe_stage` is executed — again, a stage initiated by

[5]All bounds stated in this section are expected time bounds. Analogous high probability results can be obtained by applying standard techniques.

pipe_stage has no *lparent* and knows to use the $dchild_h$ from its *uparent* (the previous stage in the same iteration).

The interesting function is StageWait, which is called when pipe_stage_wait executes and the execution is ready to advance to the next stage (i.e., dependence from the previous iteration has been satisfied). Since a stage initiated by pipe_stage_wait has both *uparent* and *lparent*, the order maintained by OM-DownFirst should use the $dchild_h$ from its *uparent*, and the order maintained by OM-RightFirst should use the $rchild_h$ from its *lparent*. However, since Cilk-P allows the execution to skip stages, identifying a stage's *lparent* requires additional work.

Consider the example shown in Figure 4. Say stage 5 of iteration i_5, denoted as (i_5, 5), had been created with pipe_stage_wait(5) instead of pipe_stage(5). Since iteration i_4 does not have a stage 5, the left parent of (i_5, 5) is (i_4, 3). Consider another example case: Say stage (i_4, 3) had been created with pipe_stage_wait(3). This would result a dependence from stage (i_3, 0) to stage (i_4, 3); however, this dependence is already subsumed by the dependences from (i_3, 0) to (i_4,0) and from (i_4,0) to (i_4, 3). In this case, (i_4, 3) does not have a *lparent* despite being created by pipe_stage_wait(3). The invariant is the following: When a stage (i, s) is initiated by pipe_stage_wait and stage (i-1, s) does not exist, (i, s)'s *lparent* is (i-1, s') where s' is the largest stage in iteration i-1 such that $s' < s$ and (i-1, s') is logically in parallel with the *uparent*, (i, s-1). Otherwise, (i, s) does not have an *lparent*.

Function FindLeftParent, called in Algorithm 4, line 15, performs the additional work needed to identify a stage's *lparent* (or lack thereof). The pseudocode for this function is not shown since it is a little complex. Briefly, for every active iteration i, we keep some metadata for the previous iteration i-1; in particular, we keep an in-order array of the stage numbers i-1 has executed so far. When FindLeftParent is called in for stage (i, s), we search this array to find the correct *lparent*, and -1 is returned if *lparent* does not exist.

Execution Time: The only additional work in Algorithm 4 (compared to Algorithm 3) is FindLeftParent — this function must be carefully implemented to minimize overhead. Consider the obvious option. We can do a binary search on the metadata array — since FindLeftParent could be called for every node and there can be T_1 nodes, this can add a $\lg k$ multiplicative overhead, leading to a time bound of $O(\lg k T_1/P + \lg k T_\infty)$, where k is the maximum array size.

Observe that if FindLeftParent is called by different stages of the same iteration, the answers returned are strictly increasing — if *lparent* of stage s is s', no subsequent stage can have an *lparent* smaller than s'. Thus, within FindLeftParent(i,s) we can search the metadata array of iteration i-1 linearly starting from the smallest stage, removing all stages smaller than s' from the array. Each item is removed at most once and the cost of the search is at most the number of items removed, allowing us to amortize the work of searches against the work of the nodes removed. However, it has the disadvantage that some calls to FindLeftParent may cost upto k. All expensive searches may happen on the span, giving us the worst case bound of $O(T_1/P + kT_\infty)$.

FindLeftParent(i, s) implements a strategy that combine the best of both worlds. Say the previous iteration (i-1) has k elements in its metadata array. We start from the smallest and look at $\lg k$ elements linearly. If we find our *lparent*, we remove all elements smaller than s and return. If not, we can remove all $\lg k$ elements we looked at, since they are clearly smaller than s. Next, we do a binary search on the rest of the metadata array to find the correct answer. Note that the cost of each search is $O(\lg k)$. In addition, if the cost was c, we removed $\Omega(c)$ elements from the metadata array. Therefore, we can amortize the work in the same manner and only incur a $\lg k$ overhead on the span, giving us the bound of $O(T_1/P + \lg k \cdot T_\infty)$ for PRacer.

Composability with Fork-Join Parallelism: Cilk-P allows programmers to compose fork-join and pipeline constructs. Each stage can itself be a series-parallel dag or a 2D dag and the nesting can be arbitrarily deep. Since nested 2D dags are also 2D dags, PRacer obviously applies directly when pipelines are nested inside pipelines. We now describe how we can handle nested fork-join parallelism.

2D-Order's SP-maintenance algorithm is similar in spirit to WSP-Order which is a parallel algorithm for detecting races in fork-join programs [60]. WSP-Order also keeps track of two total orders of the executed strands: ***English*** order and ***Hebrew*** order. The two strands are logically in parallel if and only if their relative order in English and Hebrew differ. English order is analogous to OM-DownFirst and Hebrew order is analogous to OM-RightFirst.

Nested fork-join parallelism is handled in a straightforward manner. When a stage is a series-parallel dag, we simply insert the nodes of this dag in English order in OM-DownFirst structure and in Hebrew order in OM-RightFirst structure. Series-parallel relationships are still checked by comparing relative orders of strands in the two structures. It is also straightforward to see why this is correct. Imagine a stage that was a single node u, represented by a single element in the OM data structures. When this node is replaced by a series-parallel dag G', this algorithm will replace the representative element of u in OM-DownFirst by the all the nodes in G in English order and in OM-RightFirst by nodes in G in Hebrew order. All nodes in G would have the same relationship with other nodes in the pipeline as u did.

5 Performance Evaluation

This section summarizes the implementation of PRacer and evaluates its practical performance on three benchmarks in terms of overhead and scalability. We first evaluate the performance of only the series-parallel maintenance — that is, each node is inserted into OM data structures as shown in Algorithm 4, but memory accesses are not instrumented. These experiments indicate that the overhead of 2D-Order's SP-maintenance less than 1% in all benchmarks we examined, and it provides similar scalability as the baseline program without SP-maintenance. We also evaluated the full race-detection algorithm including access histories. In this case, as with all race-detection algorithms, the overhead is significant, between 14.7–41.1× overhead compared to the baseline. However, we still get scalability similar to the baseline; therefore, some of the overhead can be offset by running race detection in parallel.

Implementation of PRacer: We implemented PRacer by extending an open-source Cilk-P runtime system, released by Intel [26], which supports the pipeline constructs as macro-defines and the corresponding work-stealing scheduler to schedule them. The implementation of PRacer consists of two components, the race detection tool component and the runtime component.

The tool component ensures that functions shown in Algorithm 4 are called at the appropriate time to perform insertions into the two OM data structures, queries are performed on memory accesses, and manages metadata for `FindLeftParent` and access histories. The tool component is called via instrumentation inserted into Cilk-P control constructs (described in Section 4.1) and memory references. We enabled the instrumentation of pipeline constructs by modifying the macros defining the pipeline constructs in Cilk-P. For memory accesses, we piggyback on the ThreadSanitizer instrumentation [55] that came with the LLVM/Clang compiler (version 3.4.1).

The runtime component required significant modification to the Cilk-P's work-stealing scheduler to allow for concurrent OM data structures based on the scheme described by Utterback et al. [60]. At a high-level, their concurrent OM data structure does ***parallel rebalances*** — occasionally, large portions of the data structure may be re-organized in parallel. The work-stealing scheduler must be designed to (1) perform appropriate concurrency control so that inserts do not occur during a parallel rebalance; and (2) appropriately move workers between the main program and the parallel rebalance. Utterback et al. implemented their system in open-source Cilk Plus runtime system released by Intel [25]. For PRacer, we re-implemented their strategy in the Cilk-P runtime system since the original runtime does not support pipelines.

Experimental Setup: We use three benchmarks to evaluate PRacer: `ferret`, `lz77`, and `x264`. Benchmark `ferret` performs content-based similarity search on images. Benchmark `lz77` is a lossless, dictionary file compression algorithm. Benchmark `x264` is an video encoder. Both `ferret` and `x264` are from PARSEC benchmark suite [7] and modified to use Cilk-P's pipeline constructs. They are both evaluated using the largest input data set, `native`, that comes

	stages / iter	*# of iters*	*# of reads*	*# writes*
`ferret`	5	3501	1.23e11	1.23e10
`lz7`	3	162	8.96e10	2.97e10
`x264`	71	36352	1.12e12	1.17e11

Figure 5. The execution characteristics of the benchmarks.

with PARSEC. We implemented `lz77` from scratch and ran it with an input text file of size 162-MBytes.

Figure 5 shows the characteristics of these benchmarks. Both `ferret` and `lz77` have relatively simple pipelines, where the structure of the pipeline is static and has a fixed number of stages across iterations, five and three respectively. On the other hand, `x264` utilizes the on-the-fly feature of Cilk-P's pipeline parallelism — even though the number of stages across iterations are the same, they can take on different stage numbers from one iteration to another.

We ran all our experiments on an Intel Xeon E5-4620 with 32 2.20-GHz cores on four sockets. Each core has a 32-KByte L1 data cache, 32-KByte L1 instruction cache, a 256-KByte L2 cache. There are a total of 500 GByte of memory, and each socket share a 16-MByte L3-cache. All benchmarks are compiled with LLVM/Clang version 3.4.1 with `-O3` running on Linux kernel version 3.10. Each data point is the average of 10 runs with standard deviation less than 5%.

Overhead of PRacer: To get a sense of PRacer's overhead breakdown, we ran the benchmarks with three different configurations: the ***baseline*** configuration, which is the the original program without race detection, the ***SP-maintenance***, which is the execution with only the SP-maintenance component of the 2D-Order without memory instrumentation; and ***full***, which is the execution with the full 2D-Order including both the SP-maintenance and access history management.

Figure 7 shows the sequential (T_1) running time for all of the three configurations.[6] The overhead due to SP-maintenance is insignificant for all benchmarks. On the other hand, adding memory instrumentation increases overheads significantly. This is explained by the fact that each stage is inserted at most twice in each OM data structure and the number of stages is relatively small (2.5e6 for `x264`). The total number of memory accesses is many orders of magnitude larger. However, these results are consistent with the overhead of full race detection in the literature [60].

Scalability of PRacer: Finally, we show that PRacer scales similarly compared to the baseline. Figure 6 shows the scalability plot of the three benchmarks. As can be seen in the plots, the scalability of the SP-maintenance and the full configurations track closely to that of the baseline. This scalability is especially useful since race detection is so expensive — serially, `x264` takes 4 hours with full race detection.

[6]The running times for `x264` are much slower than what was shown in the literature, because we disabled the vectorization code in order to perform race detection correctly.

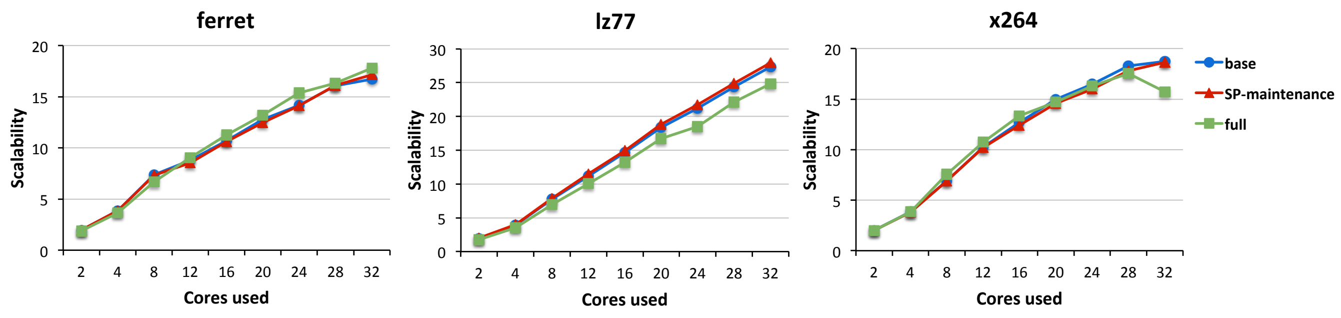

Figure 6. The scalability of the benchmarks. The x-axis shows the number of cores used. The y-axis shows the scalability, computed by taking the runtime on one core divided by the runtime on P cores under the same configuration, where P is the number cores used.

	baseline	*SP-maintenance*	*full*
ferret	191.902	191.987 (1.00×)	7984.067 (41.60×)
lz77	116.079	117.902 (1.02×)	1703.636 (14.68×)
x264	933.721	934.572 (1.00×)	15877.110 (17.00×)

Figure 7. The execution times for the benchmarks running on one core for all configurations, shown in seconds. The numbers in parentheses indicate the overhead compared to the baseline.

The parallelism of PRacer cuts this running time to a more reasonable amount for debugging.

6 Related Work

Race detection has been studied for decades since Netzer and Miller [39] formalized definitions for different kinds of races. Beyond the dynamic on-the-fly approach which we took, other approaches include static checking for races [16, 17, 36] and post-mortem analysis [2, 10, 37, 38]. There is extensive work on dynamic race detection for parallel computations that do no have well-defined structure [15, 21, 41, 45, 54, 57, 62]. Here, we focus our attention on dynamic race detection for structured programs.

As mentioned in Section 1, the race detection for fork-join program has been extensively studied [6, 18, 19, 35, 40, 60]. Most related to this paper are SP-Order [6] and WSP-Order [60]. SP-Order was the first race-detection algorithm for series-parallel dags to provide asymptotically optimal sequential running time. Analogous to this paper, it also uses two order-maintenance data structures to maintain two traversals. WSP-Order parallelizes SP-Order by designing a new runtime system and a new OM data structure — it is the first algorithm to get optimal parallel runtime. We took similar approach used in WSP-Order to allow concurrent accesses to OM data structures used by 2D-Order.

Researchers have also considered other structured programs. Lee et al. [30] considered fork-join programs that use reducers, which leads to dags more general than series-parallel dags, but still structured. Most related result to this paper is by Dimitrov et al. [14], which provides a sequential race detection algorithm for 2D-dags. Their algorithm runs sequentially and uses Tarjan's nearly linear-time least-common-ancestor (lca) algorithm [58] to identify the lowest-common-descendent of a pair of nodes in order to deduce whether they are in parallel. The use of Tarjan's lca algorithm leads to a small overhead in running time (functional inverse of Ackermann's function, which is small in practice — bounded above by 4). In contrast, 2D-Order provides a better theoretical bound sequentially; runs in parallel; and provides asymptotically optimal parallel running time.

2D-dags (or lattices, in general) have been studied extensively in graph theory. In particular, it has been known for decades that two total orders are sufficient to specify the partial order of a 2D-dag [3]. The contribution of this paper is to define the two orders that are sufficient and to show how to maintain them dynamically.

Acknowledgments

We thank the reviewers for their valuable comments and helpful suggestions. This research was supported in part by National Science Foundation under grant number CCF-1527692, CCF-1439062, CCF-1150036, and CCF-1733873.

A Artifact Evaluation

PRacer is open source and available at https://gitlab.com/wustl-pctg-pub/pracer.git. The tool library and the lz77 benchmark are provided under the MIT License. The runtime modifications are licensed under a BSD license. The modification to benchmarks ferret and x264 are released under the GNU license. The repository contains complete instructions for compiling and using PRacer, in addition to scripts that reproduce the empirical results. Please send feedback or file issues at our gitlab repository to help us continually improve the project.

References

[1] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. 2010. Executing Task Graphs Using Work-Stealing. In *24th IEEE International Parallel*

and Distributed Processing Symposium. 1–12.

[2] Todd R. Allen and David A. Padua. 1987. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the 1987 International Conference on Parallel Processing*. 721–727.

[3] K. A. Baker, P. C. Fishburn, and F. S. Roberts. 1972. Partial orders of dimension 2. *Networks* 2, 1 (1972), 11–28. https://doi.org/10.1002/net.3230020103

[4] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, 735–736.

[5] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two Simplified Algorithms for Maintaining Order in a List. In *Proceedings of the 10th European Symposium on Algorithms*. 152–164.

[6] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*. 133–144.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

[8] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. 51–61.

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 519–538.

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

[11] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. 2003. Spidle: a DSL approach to specifying streaming applications. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*. 1–17.

[12] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. 2008. Programming with exceptions in JCilk. *Science of Computer Programming* 63, 2 (Dec. 2008), 147–171.

[13] P. Dietz and D. Sleator. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. New York City, 365–372.

[14] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. 2015. Race Detection in Two Dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, Portland, Oregon, USA, 101–110.

[15] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, San Diego, California, USA, 245–255.

[16] Perry A. Emrath and Davis A. Padua. 1988. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*. 89–99.

[17] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. ACM, Bolton Landing, NY, USA, 237–252.

[18] Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. 1–11.

[19] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* (1999).

[20] Jeremy T. Fineman. 2005. *Provably Good Race Detection That Runs in Parallel*. Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

[21] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Dublin, Ireland, 121–133.

[22] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. ACM, 212–223.

[23] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 151–162.

[24] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. 2010. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 121–130.

[25] Intel Corporation. 2011. Intel® Cilk™Plus. Available from https://www.cilkplus.org/. (2011). Accessed: August 2017.

[26] Intel Corporation. 2013. Piper: Experimental Language Support for Pipeline Parallelism In Intel® Cilk™Plus. Available from https://www.cilkplus.org/piper-experimental-language-support-pipeline-parallelism-intel-cilk-plus. (2013). Accessed: August 2017.

[27] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. 2010. Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 411–420.

[28] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. 2013. On-the-Fly Pipeline Parallelism. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 140–151.

[29] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. 2015. On-the-Fly Pipeline Parallelism. *ACM Transactions on Parallel Computing* 2, 3, Article 17 (Sept. 2015), 42 pages.

[30] I-Ting Angelina Lee and Tao B. Schardl. 2015. Efficiently Detecting Races in Cilk Programs That Use Reducer Hyperobjects. In *SPAA '15: Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, Portland, Oregon, USA, 111–122.

[31] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *Journal of Supercomputing* 51, 3 (March 2010), 244–257.

[32] Steve MacDonald, Duane Szafron, and Jonathan Schaeffer. 2004. Rethinking the Pipeline as Object-Oriented States with Transformations. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments at IPDPS*. 12–21.

[33] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: a system for programming graphics hardware in a C-like language. In *ACM SIGGRAPH*. 896–907.

[34] Michael McCool, Arch D. Robison, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science.

[35] John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing'91*. 24–33.

[36] John Mellor-Crummey. 1993. 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*. ACM Press, 129–139.

[37] Barton P. Miller and Jong-Deok Choi. 1988. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 135–144.

[38] Robert H. B. Netzer and Barton P. Miller. 1989. Detecting Data Races in Parallel Program Executions. In *In Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*. MIT Press, 109–129.

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

[40] Itzhak Nudler and Larry Rudolph. 1986. Tools for the Efficient Development of Efficient Parallel Programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*.

[41] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. ACM, New York, NY, USA, 167–178.

[42] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface, Version 4.0. Available from http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. (2013).

[43] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 105–118.

[44] Antoniu Pop and Albert Cohen. 2011. A Stream-computing Extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, 5–14.

[45] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. *Concurrency and Computation: Practice and Experience* 19, 3 (March 2007), 327–340.

[46] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage Decoupled Software Pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 114–123.

[47] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer Berlin / Heidelberg, 368–383.

[48] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 531–542.

[49] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 177–188.

[50] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. 1995. Run-time Methods for Parallelizing Partially Parallel Loops. In *Proceedings of the 9th International Conference on Supercomputing*. ACM, 137–146.

[51] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. 1995. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming* 23, 6 (01 Dec. 1995), 537–576.

[52] Lawrence Rauchwerger and David A. Padua. 1999. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (Feb. 1999), 160–180.

[53] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 22–32.

[54] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*.

[55] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, 62–71.

[56] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. 2010. Feedback-directed Pipeline Parallelism. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 147–156.

[57] Rishi Surendran and Vivek Sarkar. 2016. *Dynamic Determinacy Race Detection for Task Parallelism with Futures*. Springer International Publishing, 368–385.

[58] Robert Endre Tarjan. 1979. Applications of Path Compression on Balanced Trees. *Journal of the Association for Computing Machinery* 26, 4 (October 1979), 690–715.

[59] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. 2007. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 356–369.

[60] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. 83–94.

[61] Jacobo Valdes. 1978. *Parsing Flowcharts and Series-Parallel Graphs*. Ph.D. Dissertation. Stanford University. STAN-CS-78-682.

[62] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. ACM, 221–234.