Race Detection in Two Dimensions

Dimitar Dimitrov
Department of Computer Science, ETH Zürich
Universitätstrasse 6
8092 Zürich, Switzerland
dimitar.dimitrov@inf.ethz.ch

Martin Vechev
Department of Computer Science, ETH Zürich
Universitätstrasse 6
8092 Zürich, Switzerland
martin.vechev@inf.ethz.ch

Vivek Sarkar
Department of Computer Science, Rice University
6100 Main St.,
Houston, TX, USA
vsarkar@rice.edu

ABSTRACT

Dynamic data race detection is a program analysis technique for detecting errors provoked by undesired interleavings of concurrent threads. A primary challenge when designing efficient race detection algorithms is to achieve manageable space requirements.

State of the art algorithms for unstructured parallelism require $\Theta(n)$ space per monitored memory location, where n is the total number of tasks. This is a serious drawback when analyzing programs with many tasks. In contrast, algorithms for programs with a series-parallel (SP) structure require only $\Theta(1)$ space. Unfortunately, it is currently poorly understood if there are classes of parallelism beyond SP that can also benefit from and be analyzed with $\Theta(1)$ space complexity.

In the present work, we show that structures richer than SP graphs, namely that of two-dimensional (2D) lattices, can be analyzed in $\Theta(1)$ space: a) we extend Tarjan's algorithm for finding lowest common ancestors to handle 2D lattices; b) from that extension we derive a serial algorithm for race detection that can analyze arbitrary task graphs having a 2D lattice structure; c) we present a restriction to fork-join that admits precisely the 2D lattices as task graphs (e.g., it can express pipeline parallelism).

Our work generalizes prior work on race detection, and aims to provide a deeper understanding of the interplay between structured parallelism and program analysis efficiency.

1. INTRODUCTION

Ensuring correctness of parallel programs is a notoriously difficult matter. A primary reason for this is the potential for harmful interference between concurrent threads (or tasks). In particular, a root cause for interference is the presence of data races: two conflicting accesses to the same memory location, done by two concurrent tasks. The interference caused by data races increases the number of schedules that have to be considered in ensuring program correctness. That is the case because the outcome of a data race might depend on the execution order of the two conflicting accesses.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SPAA '15 June 13-15, Portland, OR, USA
Copyright 2015 by the authors. Publication rights licensed to ACM.
ACM 978-1-4503-3588-1/15/06 ... $15.00
DOI:10.1145/2755573.2755601

Race detection challenges.

Automatic data race detection techniques are extremely valuable in detecting potentially harmful sources of concurrent interference, and therefore in ensuring that a parallel program behaves as expected. Designing precise race detection algorithms (i.e., not reporting false positives) which scale to realistic parallel programs is a very challenging problem. In particular, it is important that a race detection algorithm continues to perform well as the number of concurrently executing threads increases.

Unfortunately, state of the art race detection techniques [13] that handle arbitrary parallelism suffer from scalability issues: their memory usage is $\Theta(n)$ per monitored memory location, where n is the number of threads in the program. As n gets larger the analyzer can quickly run out of memory, or at the very least incur prohibitive slowdowns due to overwhelming memory consumption.

A principled approach to address this problem is to design race detectors which leverage the parallel structure of a program. Languages like Cilk [5], X10 [8] or Habanero [7] provide structured-parallel constructs which express task graphs of restricted shape, namely that of series-parallel (SP) graphs. Several race detection algorithms [12, 3, 18, 17] target these specific structured-parallel constructs and are able to achieve $\Theta(1)$ memory consumption per monitored location.

Key question.

The success of race detectors which achieve $\Theta(1)$ space overhead per memory location for series-parallel graphs leads to the following fundamental question:

> Are there structures richer than SP graphs which can be analyzed in a sound and precise manner with $\Theta(1)$ space overhead per memory location?

This work.

In this work we show that structures richer than SP graphs are in fact analyzable in $\Theta(1)$ space per monitored location. We present an online race detection algorithm for programs that have the parallel structure of a *two-dimensional lattice* (2D lattice) [10]. Unlike prior work, we formulate our algorithm directly in terms of the graph structure and *not* on the programming language. Decoupling structure from language constructs leads to a clearer and deeper understanding of how the algorithm works, and also of the assumptions that it rests upon. To close the loop, we introduce a restriction of the classic fork-join constructs which expresses only those

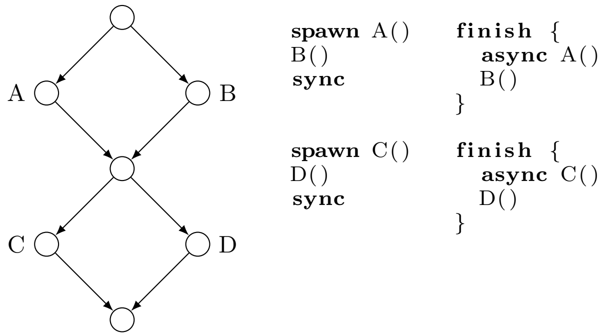


Figure 1: Two different programs having the same series-parallel task graph, one using spawn-sync while the other using async-finish.

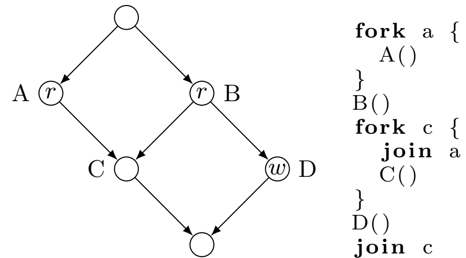


Figure 2: A fork-join program having a task graph with a two-dimensional lattice structure. Routines A and B read from the same location that D writes to. Accordingly, there is a race between A and D as they execute concurrently.

task graphs that have a 2D lattice structure. This restriction easily captures useful forms of pipeline parallelism [15], and so our race detector can be directly applied to analyze such pipelined programs. Our work can be seen as a generalization of existing race detectors for SP graphs to richer classes of graphs and language constructs.

Contributions.

The main contributions of this work are:

- an extension of Tarjan's algorithm for finding lowest common ancestors in trees, to finding suprema in 2D lattices (Tarjan's algorithm is the foundation behind prior works for SP graphs as well);
- a race detector, based on the suprema finding algorithm, that works on any task graph with a 2D lattice structure (i.e., independent of any language constructs), and has $\Theta(1)$ space overhead per tracked memory location;
- a restriction of the classic fork-join constructs which captures precisely the task graphs with a 2D lattice structure (e.g., applicable to pipeline parallelism).

Our work is a step in understanding the inherent trade-offs between parallelism structure and the resource requirements of race detection over that structure.

2. OVERVIEW

In this section we discuss several graph structures (SP graphs, 2D lattices) as well as how these are obtained from programs. We then provide an intuitive explanation of the concepts behind our online race detector for 2D lattices.

Task graphs.

Task graphs capture the ordering between operations in a particular execution of a parallel program. Operations are represented by graph vertices, and arcs (x, y) indicate that one operation y is ordered after another operation x. Thus, a task graph is a directed acyclic graph. Figures 1 and 2 show task graphs over the operations A, B, C and D, as well as some other, unnamed ones. Each of these graphs belongs to a particular class: the first is a series-parallel (SP) graph, while the second one is not SP but has the structure of a two-dimensional (2D) lattice.

2.1 SP graphs and language constructs

We now turn our attention to SP graphs, and well-known structured parallel constructs that produce such graphs. In short, an SP graph is a single-source, single-sink, directed graph which is either a series or a parallel composition of two smaller SP graphs G_1 and G_2. The serial composition $S(G_1, G_2)$ simply glues the sink of G_1 to the source of G_2, ordering the vertices in G_1 before the vertices in G_2. The parallel composition $P(G_1, G_2)$ glues the two graphs source-to-source and sink-to-sink, without imposing additional ordering on the vertices. The graph in Figure 1 can be readily constructed in this way.

Cilk's spawn-sync.

The spawn-sync parallel constructs were introduced by the Cilk [5] programming language. The **spawn** $f()$ statement activates a new parallel task to execute the given routine, while the **sync** statement suspends the currently executing task until all of its spawned children terminate. Each task has an implicit **sync** at its end. The first program in Figure 1 illustrates these two constructs. The informal semantics of spawn and sync are as follows: "**spawn** G_1; G_2" means $P(G_1, G_2)$, while "G_1; **sync**; G_2" means $S(G_1, G_2)$.

X10's async-finish.

The async-finish parallel constructs were introduced by the X10 [8] programming language, and are inherited by its descendant Habanero [7]. Here, the **async** *block* construct activates a new parallel task to execute the given block. Synchronization is done via the **finish** *block* construct, which executes the given block of code, ensuring that tasks created inside the block finish their execution together with the block. The second program in Figure 1 illustrates these constructs. An informal semantics would be: "**async** G_1; G_2" means $P(G_1, G_2)$, while "**finish** G_1; G_2" means $S(G_1, G_2)$. Note that the shown async-finish program has exactly the same task graph as the spawn-sync program to its left.

2.2 Two-dimensional lattices

In our work, we focus on task graphs which have two-dimensional (2D) lattice structure. This class of graphs is more general and extends the class of SP graphs. Two-dimensional lattices can be thought of as directed graphs that have a single source, single sink, and a monotonic planar drawing: no arcs intersect, and tracing any directed path on

the drawing will always advance in the same direction, e.g., downwards. Figure 2 shows an example of a 2D lattice task graph. The monotone-planar structure is what enables us to detect races much more efficiently.

Structured fork-join.
To express programs with a 2D lattice parallel structure, we will introduce a restricted version of the fork-join constructs. We chose **fork** and **join** because they are general enough, and with them we can naturally capture variety of other constructs such as futures. As usual, a **fork** x *block* activates a new task to execute the given block, and stores the identifier of the new task into the variable x. The **join** x statement simply suspends the current thread until the task identified by x terminates.

Figure 2 demonstrates a fork-join program with a 2D lattice task graph. In contrast with the previous two programs, here A and D execute in parallel, and thus the computation does not proceed in phases. In order to ensure two-dimensionality, we shall restrict with whom a thread may join with: if a thread y executes **join** x, then x must appear immediately on the left of y in a planar diagram of the task graph. Details are discussed in Section 5.

2.3 Online race detection

We now describe the core ideas behind our online race detection algorithm (details follow in Sections 3 and 4):

1. formulating race detection as computing suprema in an execution's task graph;
2. computing suprema efficiently by traversing a 2D lattice in a particular order;
3. showing how to obtain such traversal orders from our structured fork-join constructs.

An example of a race.
Recall that a race in a particular execution occurs when two concurrent operations access the same memory and at least one of the two is a write. An online race detector runs the program, searching for races between the current operation being executed and any of the previously executed operations. The soundness guarantee that state of the art online race detectors provide is that if the program terminates with no reported races, then indeed, the program is deterministic (from the particular input state). In addition, the detector is guaranteed to be precise up to the first reported race (later ones might be false positives).

Now, consider the program in Figure 2 together with the execution of A B C D in that order. Operations A and B read and operation D writes to the same memory location, while C is a nop. A race exists between operations A and D, which an online race detector must flag when seeing D. The race occurs because operation D conflicts with A, and the two are not ordered in the task graph. Performing the same check for B and D, we observe that a directed path connects them, so the two are ordered, and not racing.

A naive algorithm.
These observations lead to a direct method for detecting races. For every location l we track the set R of prior operations that read from l, and the set W of prior operations that wrote to l. If the current operation t reads from l, then it potentially races with an operation from $K = W$; if it writes to l, then it potentially races with an operation from $K = R \cup W$. In our case, $t = \mathrm{D}$ and $K = R \cup W = \{\mathrm{A}, \mathrm{B}\}$. When executing the current operation we simply need to check whether all potentially racing operations are ordered before it, as indicated by the task graph. Denoting with $x \sqsubset y$ that y is reachable from x, we have:

$$\text{no race between } K \text{ and } t \iff K \sqsubseteq t.$$

Race detection via suprema.
This naive algorithm, however, is prohibitively expensive both in space and time, as it suggests tracking and checking against $O(|R \cup W|)$ operations. Efficient methods for online detection represent the sets R and W indirectly, in a way guaranteeing that a race is detected if and only if a race exists. Inspired by [12, 18], we shall represent each of R and W with a single vertex in the task graph. Recall that the supremum of a set K is the unique vertex $\sup K$ such that

$$K \sqsubseteq t \iff \sup K \sqsubseteq t.$$

For the graph in Figure 2 we have that $\sup\{\mathrm{A}, \mathrm{B}\}$ equals the vertex C. From the defining property of suprema, we make the following key observation:

> To detect races it is sufficient to track $\sup R$ and $\sup W$ for every location x.

If the current operation t writes to x, then we can simply check whether both $\sup R \sqsubseteq t$ and $\sup W \sqsubseteq t$ hold, and flag a race if this is not the case. Similarly, for a read we compare against $\sup W$ only. This way we can keep track only of two vertices per location, and perform at most two reachability checks per memory access. We discuss this approach to race detection in more details in Section 4.

Finding suprema efficiently.
However, applying suprema to race detection requires the ability to compute them efficiently, or otherwise we will not benefit over the naive algorithm. This is where we leverage the structure of 2D lattices. We extend Tarjan's efficient algorithm for finding lowest common ancestors in trees to finding suprema in 2D lattices, as discussed in Section 3. This way we obtain a detector that runs in constant space per location and nearly constant time per memory access. Our extension was inspired by the SP-bags race detection algorithm [12] which implicitly applies Tarjan's algorithm to the decomposition trees of SP task graphs. The key insight of Tarjan's algorithm, and consequently of SP-bags and our extension, is to traverse the input graph in an order that is simultaneously topological, depth-first and left-to-right.

As an example, our algorithm would traverse the graph in Figure 2 in the order A B C D, but not A B D C, which is not left-to-right (nor right-to-left). To obtain an online race detection algorithm, the program execution order must match the required traversal order. A central insight from SP-bags is that for Cilk programs this can be achieved by executing the program in a serial, fork-first fashion. Similarly, this is also the case for our structured fork-join discussed in Section 5. This requirement makes the algorithm serial, but that is the price we pay for efficiency.

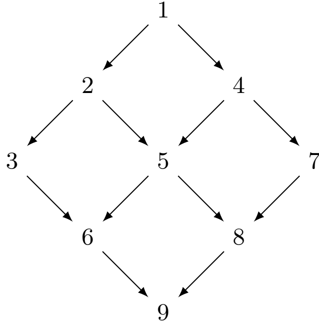
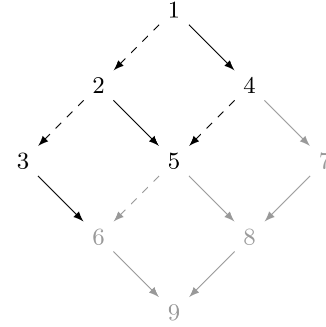


Figure 3: A planar diagram of a two-dimensional lattice. If we trace any directed path, then we always advance downwards. Arcs "intersect" only at their endpoints.



$(1,1)(1,2)(2,2)(2,3)(3,3)(3,6)(2,5)(1,4)(4,4)(4,5)(5,5)\cdots$
$\cdots(5,6)(6,6)(6,9)(5,8)(4,7)(7,7)(7,8)(8,8)(8,9)(9,9)$

Figure 4: A non-separating traversal of the arcs and the vertices of a diagram. The current point of the traversal is $(5,5)$. The black part has been "visited", while the gray part remains to be. The last-arcs are drawn solid and the rest are dashed.

3. SUPREMA IN TWO DIMENSIONS

We continue with an efficient algorithm for computing the suprema in two-dimensional lattices, a key building block for our online race detector (discussed in the next section). The algorithm takes as input a lattice diagram, and answers supremum queries on the fly while it traverses the diagram.

Lattices.

Recall that a *lattice* is a partially ordered set $(P, \sqsubset)$ such that every pair of elements $x, y \in P$ has a greatest lower bound $\inf\{x, y\}$ and a smallest upper bound $\sup\{x, y\}$, also called the *infimum* and the *supremum.* The *closure* of any subset $U \subseteq P$ is the smallest superset of U closed under infima and suprema of pairs of elements.

We represent a lattice by an acyclic digraph $G = (V, E)$ whose reachability relation is identical to $\sqsubset$. A *diagram* is a monotonic drawing of such a representation in the Euclidean plane, as shown in Figure 3. By *monotonic* we mean that tracing a directed path on the diagram will always advance in the same direction, e.g., downwards. Further, if arcs intersect at most at their endpoints, then the diagram is called *planar* and the lattice is called *two-dimensional* (Figure 3).

Non-separating traversals.

We shall traverse *both* the arcs and the vertices of the given planar diagram in a way that reveals its lattice structure, and provides us with a direct way to answer suprema queries. We formally equate a *traversal* T of the digraph $G = (V, E)$ with a permutation of $E \cup \{(x, x) \mid x \in V\}$, where each loop (x, x) represents the vertex $x \in V$.

Definition 1. A *non-separating traversal*[1] is one which is obtained by traversing a planar diagram in a topological, depth-first and left-to-right order.

Let us denote with $<_T$ the linear order in which T visits all the arcs and vertices of G. By a *topological* traversal we mean that $(a, x) \leq_T (y, b)$ whenever y is reachable from x. This implies that incoming arcs, loops, and outgoing arcs are visited in the order $(x, y) \leq_T (y, y) \leq_T (y, z)$.

A non-separating traversal of the diagram in Figure 3 is shown in Figure 4. The black part of the picture corresponds to the prefix ending in $(5,5)$. Because the traversal has to be topological, $(6,6)$ cannot be visited immediately after $(3,6)$ because we must visit $(5,6)$ before $(6,6)$.

The problem.

We are given a planar diagram of a digraph $G = (V, E)$ representing a two-dimensional lattice $(P, \sqsubset)$. We wish to traverse this diagram, and for each visited vertex $t \in V$, answer supremum queries of the form $\textsc{Sup}(x, t)$. We shall impose the following *precondition* on all queries $\textsc{Sup}(x, t)$

$$x \text{ is in the closure of the traversal prefix ending in } t. \quad (1)$$

For example, given the traversal in Figure 4 the query $\textsc{Sup}(6, 5)$ is valid, while $\textsc{Sup}(7, 5)$ is not.

For a fixed vertex t, let us collect all queries of the form $\textsc{Sup}(x, t)$ into the set $Q(t)$. Then, our task is to process the sequence of such query sets, one per every visited vertex:

$$Q(t_1), \ldots, Q(t_n) \quad (2)$$

This setting is adequate when the i-th query set $Q(t_i)$ is not given in advance to the traversal, but is determined on the fly from the prefix ending in t_i (as is the case in race detection).

Connection via forests.

We connect a non-separating traversal T to suprema via certain forests associated with the prefixes of T. Consider a fixed vertex $x \in V$ and the last visited arc[2] $(x, y) \in T$ that exits x. We refer to it as the *last arc* of x, or just as a *last-arc.* Taken together, the last-arcs form a tree directed towards its root (Figure 4). Thus, the last-arcs that belong to any given prefix of T form a forest:

Definition 2. For any traversal T and arc $(s, t) \in T$, define the *last-arc forest* $T/(s, t)$ to be the collection of all last-arcs $(x, y) \leq_T (s, t)$ belonging to the prefix ending in (s, t). The forest vertices are those incident to some arc in the forest.

[1]The name comes from the non-separating linear extensions of lattice orders, defined by Dushnik and Miller [10].

[2]Equivalently, a last-arc (x, y) is the right-most arc exiting the vertex x.

```
WALK(T, Q)
1  for (s, t) ∈ T
2      if (s, t) is a loop
3          t.visited ← TRUE
4          answer Q(t)
5      if (s, t) is a last-arc
6          UNION(t, s)
```

```
SUP(x, t)
1  r ← FIND(x)
2  if r.visited
3      return t
4  else
5      return r
```

Figure 5: An algorithm for finding suprema in two-dimensional lattices. The input lattice is encoded by a non-separating traversal T. The algorithm answers queries $\text{Sup}(x,t) \in Q(t)$. In practice, Q can be thought of as a callback invoking Sup.

We think of (s,t) as the "current" point in the traversal, splitting it into visited prefix and unvisited suffix. For the traversal in Figure 4, the current point is $(5,5)$, and the forest $T/(5,5)$ is the union of the trees $\{(3,6)\}$, $\{(2,5)\}$ and $\{(1,4)\}$. Note that vertex 6 belongs both to the closure of the visited prefix and also to the forest $T/(5,5)$. In general, the closure of the prefix ending in (t,t) always equals the vertices of the forest $T/(t,t)$.

The connection between a non-separating traversal T and suprema goes through the roots of the forest $T/(t,t)$ as described by the following

Theorem 1. *Given a non-separating traversal T and a pair of vertices x and t, where x belongs to the closure of prefix ending in (t,t), let r be the root of the tree in $T/(t,t)$ that contains x. Then $\sup\{x,t\}$ attains the form:*

$$\sup\{x,t\} = \begin{cases} t & \text{if } r \leq_T t \\ r & \text{if } t \leq_T r. \end{cases}$$

If on Figure 4 we let $x = 3$ and $t = 5$, then $r = 6$. Vertex 6 is traversed after 5, and so $\sup\{x,t\}$ equals vertex 6. On the other hand, if $x = 1$ and $t = 5$, then $r = 4$ and $\sup\{x,t\}$ equals vertex 5. We shall prove Theorem 1 in Section 6.

Suprema finding algorithm.

Theorem 1 leads directly to an algorithm for answering supremum queries. Perform a non-separating traversal T, and maintain the forest $T/(s,t)$ at every point $(s,t) \in T$. When at a vertex t, answer all queries of the form $\text{Sup}(x,t)$:

Find the root r of the tree containing x; if the root has been visited, then answer t; otherwise answer r.

Figure 5 lists a pseudo-code for this algorithm. The main routine Walk performs the traversal and maintains the last-arc forest, while the subroutine Sup answers individual queries. The main routine accepts the traversal T directly, and also a description Q of all queries. The subroutine then answers each query $\text{Sup}(x,t) \in Q(t)$. The algorithm uses an union-find data structure to maintain the mapping from each visited vertex to its root in the last-arc forest: the vertices of each tree are kept in a disjoint set labeled by the root of the tree. The $\text{Find}(x)$ operation returns the label of the set containing x. The $\text{Union}(y,x)$ operation merges the sets containing y and x under the label of the set containing y. Initially, every vertex x is alone in a set $\{x\}$ labeled by x.

Theorem 2. *The algorithm in Figure 5 is correct.*

Proof. By Theorem 1 it is sufficient to argue that we maintain the vertices of each tree in their own set labeled by the tree root. Assume this holds for the step $(x,y) \in T$ before we visit a last-arc $(s,t) \in T$. Because T is topological both s and t are roots in $T/(x,y)$. The new forest $T/(s,t)$ differs by having s attached as a child of t. This is exactly what lines 5–6 of the main routine Walk accomplish. □

We now elaborate on the algorithm's resource requirements. An union-find data structure can be implemented very efficiently, guaranteeing nearly constant amortized time per operation. The precise asymptotics is given in terms of Tarjan's functional inverse α of the Ackermann function.

Theorem 3. *The algorithm in Figure 5 needs at worst $\Theta\big((m+n)\alpha(m+n,n)\big)$ time and $\Theta(n)$ space to answer m supremum queries on a lattice with n elements.*

Proof. In total at most $m+n$ union-find operations are executed over n elements: one Find per query and at most one Union per element. With a fast union-find implementation they shall take at most $\Theta\big((m+n)\alpha(m+n,n)\big)$ time, as analyzed by Tarjan [19, 20], and $\Theta(n)$ space. Additionally, by Euler's formula at most $3n-6=\Theta(n)$ arcs are traversed, as the input diagram is planar. □

Remark 1. We assumed that a planar diagram or a non-separating traversal are directly given as input. Therefore, there stands the question of whether we can obtain them efficiently in the context of online race detection. In Section 5 we discuss a restriction to fork-join for which this is the case.

In a more general context, it is worth recalling how to obtain a planar diagram or a non-separating traversal given the input digraph alone. Without loss of generality, we can consider graphs with a single source s, a single sink t, and an arc (s,t) connecting the two. For such digraphs, we can obtain a *monotonic* planar drawing (or a non-separating traversal) from a *any* planar drawing where the arc (s,t) lies on the external face [2, 14]. Planar drawings can be constructed efficiently in linear time, e.g., by [6, 9].

Remark 2. The algorithm we presented can be seen as an extension of Tarjan's offline algorithm for finding lowest common ancestors in trees. A lowest common ancestor is just another name for infimum, and by reversing the arcs in a directed graph (poset) we switch infima and suprema. Thus, we can see Tarjan's algorithm as finding suprema in a semilattice with the shape of a tree. In this case, a simpler version of Theorem 1 holds:

For the root r of the tree in $T/(t,t)$ containing x,
it is always the case that $t \leq_T r$, and therefore:

$$\sup\{x,t\} = r.$$

That is, in this case we do not need to track whether the root has already been visited or not.

Our extension to Tarjan's algorithm is inspired by the SP-bags algorithm due to Feng and Leiserson [12]. SP-bags basically applies Tarjan's algorithm to the decomposition tree of a series-parallel graph, which can be shown to be equivalent to applying Theorem 1 to the series-parallel graph directly.

Next, we show how our algorithm can be used as a building block in an online race detector.

4. ONLINE RACE DETECTION

In this section we consider the problem of detecting races in programs whose task graphs have a two-dimensional lattice structure. In particular, we apply the algorithm for finding suprema from Section 3 in an efficient race detection algorithm. Interestingly, as we discuss below, non-separating traversals are not always obtainable in the context of online race detection (as these traversals may consist of events that have not yet occurred in an execution). Thus, we show how to slightly adapt the class of traversals we consider in order to achieve a fully online algorithm for 2D lattices.

Races.

Recall that given a program execution, a *race* between a pair of conflicting memory operations exists if these operations are not synchronized to occur in a fixed order. Two operations conflict whenever they access the same memory location, and at least one of them writes to that location. The presence of racing operations indicates potentially non-deterministic behavior, as executing them in different orders might lead to different results.

We reason about operation order by querying the task graph $G = (V, E)$ of the particular execution that we are analyzing. An operation x is ordered before y, if from x we can reach y via a directed path in G. The execution itself corresponds to a traversal of the graph in topological order. A race manifests as two vertices not connected by a directed path and whose corresponding operations do conflict. To keep the graph-theoretic terminology we shall refer to operations as vertices.

The problem.

We are given a program such that the task graph of any of its executions has a two-dimensional (2D) lattice structure, i.e., a planar diagram. Our goal is to detect races online as we execute the program. In abstract terms, the execution produces a task graph G and a traversal T that we follow. For each visited vertex t, we wish to detect whether t races with another visited vertex $x <_T t$. We shall assume that T has a certain structure, namely, be non-separating. Finding an execution order which ensures that structure depends on the class of programs under consideration. In Section 5 we discuss a structured use of fork-join for which an appropriate execution order can be easily determined.

Suprema based race detector.

We arrive at a race detector by applying the algorithm for answering supremum queries from Section 3 to the lattice $(P, \sqsubset)$ determined by the task graph (recall that $\sqsubset$ equals the reachability relation of the task graph G). Of course, the traversal T dictated by the program execution must be non-separating. While traversing the task graph along T, we consider the set K of visited vertices that conflict with the current vertex t. Unless every vertex in K is ordered before t we have a race:

$$\text{no } K\text{-}t \text{ race} \iff K \sqsubseteq t \iff \sup K \sqsubseteq t. \tag{3}$$

We can therefore reduce race detection to tracking suprema in the lattice $(P, \sqsubset)$. Note that $\sup K$ need not even access the same memory location as K and t (cf. Figure 2).

The overall approach is summarized in Figure 6. Two routines On-Read and On-Write handle respectively read

```
On-Read(t)
1  if Sup(R[t.loc], t) ≠ t
2       report a race on t.loc
3  R[t.loc] ← Sup(R[t.loc], t)

On-Write(t)
1  if Sup(R[t.loc], t) ≠ t or Sup(W[t.loc], t) ≠ t
2       report a race on t.loc
3  W[t.loc] ← Sup(W[t.loc], t)
```

Figure 6: Online race detection via suprema. For every memory operation t in an program execution a corresponding routine is performed. The maps $R[loc]$ and $W[loc]$ accumulate the suprema of respectively all reads and writes per location loc.

and write operations. For every memory location loc they accumulate into $R[loc]$ the supremum of those visited vertices that read from loc, and accumulate into $W[loc]$ the supremum of those visited vertices that write to loc. To check for races on the location $t.loc$ accessed by the current vertex t, the routines compare $R[t.loc]$ or $W[t.loc]$ with t in the lattice order $\sqsubset$. Such comparison $x \sqsubseteq t$ is implemented with the query $\textsc{Sup}(x, t) = t$. The suprema $R[t.loc]$ and $W[t.loc]$ are always in the closure of the vertices visited up to t as required by the problem definition in Section 3.

Obstacles to online race detection.

In general, a program may not have an execution that corresponds to a non-separating traversal, even though its task graphs have a two-dimensional lattice structure. That is because a non-separating traversal might require us to visit an arc (s, t) at a moment where t is *not yet* determined by the execution so far. For example, suppose that (s, t) is the only arc exiting s, the vertex s is the last operation of some thread, and t is a **join** operation joining that thread. Then, the presence of the arc (s, t) is determined only upon the execution of t, while a non-separating traversal requires that (s, t) is visited right after (s, s). A similar difficulty arises in answering supremum queries: when visiting a vertex t the supremum $\sup\{x, t\}$ might be indeterminate as well. These cases are demonstrated by Figure 2 in Section 2.

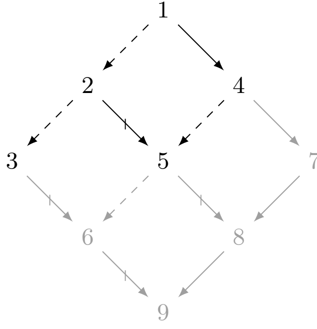
In order to obtain an online race detector, we shall do two things: 1) employ a slightly different class of traversals that have no obstacle to being executable; 2) relax the problem of finding suprema, such that it can be solved over the new class of traversals, and moreover still facilitate race detection.

Delayed traversals.

We shall now define *delayed* non-separating traversals. First, let us characterize the arcs (s, t) that can potentially prevent a traversal T from being executable. Assume that T visits (s, t) before it visits a vertex x on which the vertex t depends on, i.e., assume that

$$(s, t) <_T (x, x) \text{ and } x \sqsubset t. \tag{4}$$

Whether the arc (s, t) is present in the task graph is usually determined right before the execution of t, and therefore only after the execution of x. In this case no execution corresponds to T. An example of (4) is given by $(3, 6) <_T 5 \sqsubset 6$ in Figure 4, Section 3.



$(1,1)\cdots(3,3)(3,\times)(2,\times)(1,4)(4,4)(2,5)(4,5)(5,5)\cdots$

Figure 7: A delayed non-separating traversal. The traversal of the crossed arcs has been delayed so they are visited together with their target vertices. The stop-arcs $(3,\times)$, $(2,\times)$, etc. (not drawn) mark the original places of the delayed arcs.

To remove this obstacle, we shall *delay* the traversal of all arcs (4) until immediately before t. Hence, in the old place of (s,t) we leave the special marker $(s,\times)$ that we call a *stop-arc*. We obtain the transformation $T \mapsto T'$:

$$
\begin{array}{llllllll}
T: & \cdot\cdot & (s_i,t) & \cdot\cdot & (s_j,t) & \cdot\cdot & & (s_n,t)(t,t) \\
 & & \downarrow & & \downarrow & & & \downarrow \\
T': & \cdot\cdot & \underbrace{(s_i,\times)}_{\text{stop-arc}} & \cdot\cdot & \underbrace{(s_j,\times)}_{\text{stop-arc}} & \cdot\cdot & \underbrace{\cdot\cdot(s_i,t)\cdot\cdot(s_j,t)\cdot\cdot}_{\text{delayed arcs}} & (s_n,t)(t,t)
\end{array}
$$

Definition 3. A *delayed non-separating traversal* is one which is obtained in the same way as a non-separating traversal except that the arcs (4) have been delayed and stop-arcs mark their original places (Figure 7).

In Section 5 we discuss a restriction to fork-join for which delayed traversals can easily be obtained.

Relaxed query problem.

As $\sup\{x,t\}$ might be indeterminate at the moment of execution of t, we shall relax the original query problem (2) from Section 3. Observe that the race detection algorithm in Figure 6 uses the result of a query only to compare it with the current vertex. Therefore, we may answer queries differently as long as any such sequence of comparisons leads to the same outcome. Recall that the defining property of suprema is given by

$$\sup\{x,y\} \sqsubseteq t \iff x \sqsubseteq t, y \sqsubseteq t, \tag{5}$$

for all x, y and t. It is therefore sufficient to come up with a routine $\textsc{Sup}(x,t)$ that answers the relaxed query problem

$$\textsc{Sup}(x,t) = t \iff x \sqsubseteq t \tag{6}$$

$$\textsc{Sup}(\textsc{Sup}(x,y),t) = t \iff \textsc{Sup}(x,t) = t, \textsc{Sup}(y,t) = t \tag{7}$$

for all vertices x, y and t that satisfy the precondition (1) to $\textsc{Sup}$ from Section 3. For example, if we execute the program in Figure 2, Section 2 in the order A B C D, then $\textsc{Sup}$(A, B) is allowed return A instead of the true supremum C.

The condition (5) and the conditions (6)-(7) are of course not equivalent, for otherwise $\textsc{Sup}(x,y)$ must always equal $\sup\{x,y\}$. The difference is that for (6)-(7) the possible combinations of x, y and t are restricted by the $\textsc{Sup}$ precondition (1), while for (5) they are unrestricted.

```
Walk(T, Q)
1 for (s, t) ∈ T
2     if (s, t) is a loop
3         t.visited ← True
4         answer Q(t)
5     if (s, t) is a last-arc
6         Union(t, s)
7     if (s, t) is a stop-arc
8         s.visited ← False
```

```
Sup(x, t)
1 r ← Find(x)
2 if r.visited
3     return t
4 else
5     return r
```

Figure 8: An algorithm for answering relaxed supremum queries (6)-(7) along delayed non-separating traversals. The only difference with the algorithm in Figure 5 is that this one handles stop-arcs $(s,\times)$ by marking the vertex s as unvisited.

Modified algorithm.

We shall adapt the algorithm in Figure 5, Section 3 to solve the relaxed query problem (6)-(7) along delayed non-separating traversals. In answering queries $\textsc{Sup}(x,t)$, we need to decide what to do when the supremum $s = \sup\{x,t\}$ has *not* been visited yet. Recall that along a normal non-separating traversal T, we can simply find the root of x in the forest $T/(t,t)$. Because s is not visited, by Theorem 1 this root must equal s. However, in the corresponding delayed traversal T' the root r of tree in $T'/(t,t)$ that contains x does not equal s, but merely has the last-arc (r,s) pointing to s. Because the last-arc (r,s) has been delayed after t, at this point we do not know what the true supremum is.

To deal with this situation, we shall answer such queries with $\textsc{Sup}(x,t) = r$, pretending that r is the supremum s. We need to make sure that r is marked as unvisited, so in future queries it behaves the same way as s does. When we later visit the arc (r,s) we have the chance to correct this deception by attaching r as a child of s, so that no one will notice. The moment when r should start acting like s, is when we visit the stop-arc $(r,\times)$. Then, we mark r as unvisited. Recall that the stop-arc stands in the original place of the last-arc (r,s) in the traversal T.

In essence, by marking the root r as unvisited, we make it observationally equivalent to the supremum s with respect to (6)-(7). This approach is summarized in Figure 8.

THEOREM 4. *The algorithm in Figure 8 is correct with respect to the relaxed conditions (6)-(7).*

PROOF. We begin with condition (6). Recall that along a non-separating traversal T the forest $T/(s,t)$ changes with every visited last-arc (s,t) by attaching the root s as a child of the root t. In the corresponding delayed traversal T' exactly the same changes occur but at the point right before t is visited. Consider the state of the algorithm along T' and compare the two forests $T/(t,t)$ and $T'/(t,t)$. The trees in $T/(t,t)$ having roots visited before t are precisely the trees in $T'/(t,t)$ having roots marked as visited. Therefore, by Theorem 1 we obtain (6). As for condition (7), compare the answer $r = \textsc{Sup}(x,y)$ over T' and $s = \sup\{x,y\}$. By the definition of a delayed traversal, (r,s) must be a last-arc. Moreover, this arc is visited before t, and so r must be attached as a child of s before t is visited. Then, assuming either side of the equivalence (7), the vertices x, y and t all belong to the same tree in $T'/(t,t)$, with t being its root. Therefore, the other side of (7) follows. □

Space and time and requirements.

We now turn to the question of resource requirements. As currently formulated, the algorithm requires storing every visited vertex, i.e., requires at minimum space proportional to the number of executed operations. This can be too expensive in practice, and we now discuss how to reduce this number significantly. The idea is to decompose the vertices of the task graph into "threads" and identify each vertex with the thread that it belongs to, while ensuring that the race detection algorithm is sound and precise. This way we need to store only the threads and not the vertices themselves.

For a given delayed traversal T' of the graph $G = (V, E)$, we define a *thread* as the set of vertices of a maximal path of non-delayed last-arcs. For example, the threads in Figure 7 are $\{2\}$, $\{3\}$, $\{5\}$, $\{6\}$, and $\{1, 4, 7, 8, 9\}$. We assume that each thread is assigned an unique identifier, and let $tid(x)$ denote the identifier of the thread containing x. Instead of feeding the delayed traversal T' to the race detector, we transform it $T' \mapsto T''$ by replacing every arc[3] according to

$$(x, y) \mapsto (tid(x), tid(y)). \quad (8)$$

This way the race detector operates on threads instead of vertices, and does bookkeeping proportional to the number of threads. Moreover, the transformation preserves every comparison made by the race detection:

$$\text{SUP}(x, t) = t \iff \text{SUP}(tid(x), tid(t)) = tid(t). \quad (9)$$

This follows from the fact that a thread intersects at most one tree in $T'/(s, t)$ for every $(s, t) \in T'$.

From the already calculated resource bounds of the algorithm in Section 3 (i.e., Theorem 3), we directly obtain:

THEOREM 5. *The race detection algorithm in Figure 6 needs $\Theta(\alpha(m + n, n))$ amortized time per executed operation, where m is the number of operations, and n is the number of threads. Also, it needs $\Theta(1)$ space per thread and per tracked memory location.*

For the structured fork-join program constructs, discussed in the next section, the threads defined here correspond to actual program threads (i.e., tasks).

5. STRUCTURED FORK-JOIN

In this section we restrict the fork-join parallel constructs, such that they produce the task graphs with a two-dimensional lattice structure. For the sake of presentation we will deal with task graphs in which every vertex has at most two incoming arcs, and at most two outgoing arcs. The general case is easily obtainable from this one.

Structured fork-join.

We shall structure the use of fork and join constructs by restricting with which tasks a given task is allowed to join. The basic idea is to maintain all running tasks *as points in a line*. Each task may join only its left neighbor, removing it along the way. Similarly, a newly forked child becomes the left neighbor of the parent. This way each task x splits the line into a left part L and a right part R, or more graphically into $L \cdot x \cdot R$. The restrictions mean that task x may add and remove tasks only at the right end of L (treat it like a LIFO stack), but cannot touch R at all.

[3]We assume that the information about what operation corresponds to a vertex is preserved somewhere else.

$$L \cdot \{x \mid \textbf{fork}\ y\ \beta; \alpha\} \cdot R \longrightarrow L \cdot \{y \mid \beta\} \cdot \{x \mid \alpha\} \cdot R$$
$$L \cdot \{y \mid\} \cdot \{x \mid \textbf{join}\ y; \alpha\} \cdot R \longrightarrow L \cdot \{x \mid \alpha\} \cdot R$$

Figure 9: Fork-join rules that capture task graphs with a 2D lattice structure. Each task is represented by a pair $\{x \mid \alpha\}$, where x is the task identifier and α is a list of statements. All tasks are organized as points in a line. A forked task goes on the left of its parent, and a task may only join the one on its left.

We state these rules more formally in Figure 9. Each task is represented as a pair $\{x \mid \alpha\}$, where x is an unique identifier, and α is the sequence of statements that the task executes. The program in Figure 2, Section 2 follows the rules. Figure 10 shows a 2D task graph along with lines of task points at various moments in the execution.

Vertices in a task graph correspond to transitions taken by the program, e.g, **fork** y or **join** y. Edges signify immediate dependencies of one transition upon another, e.g., a **join** y transition depends on the previous transition by the same thread and also on the final transition by the joined thread. (This is essentially Lamport's happened-before relation.)

THEOREM 6. *The rules in Figure 9 generate task graphs with a two-dimensional lattice structure.*

PROOF. From an execution that follows the rules we can easily construct a planar diagram of the task graph. Recall that a diagram is required to be monotonic, i.e., that directed paths always advance in a fixed direction. Let us choose this direction to be downwards. Now, each program transition transforms $T_i \mapsto T_{i+1}$ the current line T_i of task points, and so we have a history of line snapshots $T_1, \ldots, T_n$.

Let us lay out the lines such that each T_i is horizontal and placed above T_{i+1} (Figure 10). Observe that every task x intersects each line T_i at exactly zero or one points $x_i \in T_i$. To build the task graph, for each $T_i \mapsto T_{i+1}$ add arcs between:

1. x_i and x_{i+1},
2. x_i and y_{i+1} if x forks y
3. y_i and x_{i+1} if x joins y

and then collapse all task points that represent the same task state. The resulting diagram is monotonic as T_i is above T_{i+1}. It is also planar, because when forking or joining the two involved points $x_i, y_i \in T_i$ are always next to each other. The resulting diagram is a diagram of a lattice by a well-know result [1, 11] in poset dimension theory. □

An extension of the rules with forking and joining any number of tasks would capture all possible 2D lattices.

Obtaining delayed traversals.

The proof of Theorem 6 indicates a direct way to obtain a delayed non-separating traversal from an execution. On the resulting diagram (Figure 10) a newly forked task stays on the left of its parent task. Therefore, to traverse the diagram from left to right, we can simply execute the program *serially, fork-first*, and emit arcs on the way. Here, x and y designate task identifiers, according to (8), Section 4:

$$T \xmapsto{x\ \textbf{forks}\ y} T \cdot (x, y), \qquad T \xmapsto{x\ \textbf{steps}} T \cdot (x, x),$$
$$T \xmapsto{x\ \textbf{joins}\ y} T \cdot (y, x), \qquad T \xmapsto{x\ \textbf{halts}} T \cdot (x, \times),$$

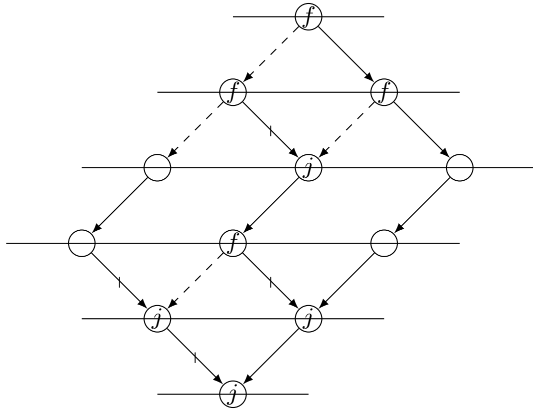


Figure 10: A fork-join task graph with a 2D lattice structure. Fork edges are dashed, step edges are solid, and join edges are crossed. A serial fork-first execution corresponds to a delayed non-separating traversal of the task graph. Lines of task points from the fork-join rules are drawn horizontally.

To motivate this construction, observe that a last-arc in the task graph connects either two consecutive operations on the same task, or a final operation on one task and a join operation from another. The one between a final operation and a join must be delayed, which is done by emitting a stop-arc when a task halts, and a last-arc when one task joins another.

To instantiate our race detection algorithm, we simply need to stream the constructed traversal T on the fly to the WALK routine in Figure 8, with the race detector in Figure 6 passed as the callback Q.

Generalization of series-parallel constructs.

It is instructive to understand how the structured fork and join presented here generalize constructs that produce series-parallel graphs, such as Cilk's spawn-sync or X10's async-finish. Consider a stylized version of the rules in Figure 9, where for clarity, program statements are omitted and only the task identifiers are kept:

$$L \cdot x \cdot R \underset{x \textbf{ joins } y}{\overset{x \textbf{ forks } y}{\rightleftharpoons}} L \cdot y \cdot x \cdot R. \tag{10}$$

A situation that produces non-SP graphs can arise here. Consider which task could have forked y. As a newly forked task is placed on the left of its parent, the task that have forked y must be either x or some task in R. For example, we can have the passage

$$t \xrightarrow{t \textbf{ forks } y} y \cdot t \xrightarrow{t \textbf{ forks } x} y \cdot x \cdot t \xrightarrow{x \textbf{ joins } y} x \cdot t.$$

This results in a non-SP task graph (cf. Figure 2, Section 2). One way to ensure that the produced task graph is SP, is to require y to be a descendant of x. This is achieved by bracketing x in (10) together with its descendants S:

$$L \cdot [S \cdot x] \cdot R \underset{x \textbf{ joins } y}{\overset{x \textbf{ forks } y}{\rightleftharpoons}} L \cdot [S \cdot [y] \cdot x] \cdot R. \tag{11}$$

This way x cannot join a task outside S. It is easy to establish that (11) indeed produces series-parallel task graphs, e.g, we recover the semantics of **sync** by automatically joining with the whole set S in addition to y.

Handling pipeline parallelism.

Before we conclude this section, we discuss another setting which can benefit from our race detector. Many applications exhibit parallel structure in the form of a linear pipeline as described in [16]: they take as input a sequence of data items $x_1, \ldots, x_n$, and feed each item x_j through a sequence of computation stages $S_1(x_j), \ldots, S_m(x_j)$.

A task $S_i(x_j)$ is allowed to depend on any $S_k(x_l)$ where $k < i$ or $l < i$, but otherwise tasks are run in parallel. Thus, the task graph of a linear pipeline can be embedded into a two-dimensional grid, i.e., it forms a two-dimensional lattice. This pattern can be directly captured in our structured fork-join and can also be analyzed with the race detection algorithm presented in this paper.

Blelloch and Reid-Miller [4] made the observation that many pipelined programs are more naturally expressed in a fork-join fashion. However, their model is more relaxed than ours as it allows non-linear pipelines, therefore leaving open the question for efficient race detection in their case. Linear pipelines are the focus of the work of Lee et al. [15] which extends Cilk with support for this setting. Interestingly, their language constructs are easily expressible in our restricted fork-join, but not the other way around, even though both models can express exactly the same task graphs, i.e., the ones having a two-dimensional lattice structure.

6. PROOFS

In this section we prove Theorem 1 in a series of four lemmas, which are basically weaker versions of it.

Let $(P, \sqsubset)$ be a two-dimensional lattice represented by a given planar diagram with digraph $G = (V, E)$, and let T be a non-separating traversal of the diagram. Without loss of generality we assume that $P = V$, and that the diagram is monotonic in the downwards direction, and that the traversal is from left to right. Recall that x and y are comparable if either $x \sqsubseteq y$ or $y \sqsubseteq x$, i.e., they lie on a directed path in G.

LEMMA 1. *Let x and t be two incomparable vertices such that x belongs to $T/(t,t)$. Then,* $\sup\{x,t\}$ *is reachable from x via a directed path consisting of last-arcs only.*

PROOF. Because the traversal is depth-first left-to-right, we can choose the planar diagram such that the vertical line crossing $\sup\{x,t\}$ has x on its left and t on its right. The diagram is planar and monotonic, hence we can select the rightmost path exiting x and the leftmost path exiting t. Then, $\sup\{x,t\}$ must lie on the intersection of the two paths, or it would not be the least upper bound of $\{x,t\}$. Any rightmost path by definition consists of last-arcs only. □

LEMMA 2. *For all vertices x and t such that x belongs to $T/(t,t)$ we have that* $\sup\{x,t\}$ *also belongs to $T/(t,t)$.*

PROOF. If x and t are comparable then the statement is trivial, so let us assume that x and t are incomparable, and let $s = \sup\{x,t\}$. Select the last-arc (p,s) lying on the path from Lemma 1 that connects x and s. The two vertices p and t must be incomparable, and therefore (p,s) must be visited before t, for otherwise the traversal would not be depth-first left-to-right. We conclude that the last-arc (p,s) belongs to the forest $T/(t,t)$ and so does s. □

LEMMA 3. *For all vertices x and t such that x belongs to $T/(t,t)$ we have that* $\sup\{x,t\}$ *is a root of $T/(t,t)$.*

PROOF. From Lemma 2 we known that $s = \sup\{x, t\}$ belongs to $T/(t, t)$. Because $T/(t, t)$ consists of last-arcs only, if s is not a root, then its last-arc must also belong to $T/(t, t)$, and therefore be visited before t. This contradicts the assumption that the traversal is topological as by the definition of supremum $t \sqsubseteq s$. □

LEMMA 4. *For every vertex t and root r in $T/(t, t)$ we have that r is comparable with t.*

PROOF. Assume r and t are incomparable. By Lemma 1 we conclude that $s = \sup\{r, t\}$ must be reachable from either r or t via a directed path of last-arcs. But by Lemma 3 we know that s is a root of the last-arc forest $T/(t, t)$, and therefore r and t cannot both be roots, a contradiction. □

THEOREM 1. *Given a non-separating traversal T and a pair of vertices x and t, where x belongs to the closure of prefix ending in (t, t), let r be the root of the tree in $T/(t, t)$ that contains x. Then $\sup\{x, t\}$ attains the form:*

$$\sup\{x, t\} = \begin{cases} t & \text{if } r \leq_T t \\ r & \text{if } t \leq_T r. \end{cases}$$

PROOF. It is not difficult to see that the set of vertices of $T/(t, t)$ equals the closure of the prefix ending in t. Now, by Lemma 4 and because the traversal is topological, the following equivalences hold (recall that $<_T$ is a linear order):

$$r \leq_T t \iff r \sqsubseteq t \tag{12}$$

$$t <_T r \iff t \sqsubset r. \tag{13}$$

If $r \leq_T t$, then the theorem follows directly, and so let us consider the case when $t <_T r$ and $t \sqsubset r$. Then, x and t are incomparable, and by Lemma 1 we have that $s = \sup\{x, t\}$ is reachable from x by a path of last-arcs. But by Lemma 3 s is a root in $T/(t, t)$, and therefore must equal r. □

Remark 3. Introducing two dimensional lattices as those having planar diagrams is intuitive. However, the original (and more flexible) definition due to Dushnik and Miller [10] describes them as those being the intersection of two linear orders. The fact that this is equivalent to having a planar diagram was proved by Baker et al. [1].

7. CONCLUSION

We presented an online algorithm for race detection in task graphs with a two-dimensional (2D) lattice structure. These 2D lattices are richer than SP-graphs and thus our algorithm generalizes race detectors for SP-graphs. Our algorithm is founded in reducing race detection to finding suprema in program task graphs. Building on prior work [12, 18], we extended Tarjan's algorithm for finding lowest common ancestors in trees, to finding suprema in 2D lattices. Efficient computation of suprema is based on performing a *non-separating* traversal of the lattice graph. However, such a traversal need not correspond to any program execution, thus precluding an online algorithm. To overcome this obstacle, we observed that a race detector can pose suprema queries with a relaxed semantics. We adopted our suprema finding algorithm to answer the relaxed queries over a wider class of traversals, and obtained a fully online race detector. Finally, we introduced restricted fork-join constructs which permit only task graphs with a 2D lattice structure, resulting in programs directly analyzable by our online algorithm.

8. ACKNOWLEDGMENTS

We thank Raghavan Raman and Boris Peltekov for helpful discussions on earlier versions of this work. We are also grateful to the anonymous reviewers for their questions and well-placed remarks which improved the paper.

9. REFERENCES

[1] K. A. Baker, P. C. Fishburn, and F. S. Roberts. Partial orders of dimension 2. *Networks*, 2(1):11–28, 1972.
[2] G. D. Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoretical Computer Science*, 61(2-3):175 – 198, 1988.
[3] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. SPAA, pages 133–144. ACM, 2004.
[4] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. SPAA, New York, NY, USA, 1997. ACM.
[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. PPOPP, 1995.
[6] J. Boyer, P. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. Graph Drawing '04. Springer, 2004.
[7] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The new adventures of old X10. PPPJ, 2011.
[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. OOPSLA, 2005.
[9] H. de Fraysseix and P. O. de Mendez. Trémaux trees and planarity. *E. J. Comb.*, 33(3):279 – 293, 2012.
[10] B. Dushnik and E. W. Miller. Partially ordered sets. *American Journal of Mathematics*, 63(3):600–610, 1941.
[11] S. Felsner, W. T. Trotter, and V. Wiechert. The dimension of posets with planar cover graphs. *Graphs and Combinatorics*, pages 1–13, 2014.
[12] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. SPAA, 1997.
[13] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. PLDI, 2009.
[14] D. Kelly. Fundamentals of planar ordered sets. *Discrete Mathematics*, 63(2-3):197 – 216, 1987.
[15] I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-fly pipeline parallelism. SPAA. ACM, 2013.
[16] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.
[17] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. PLDI, 2012.
[18] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. RV, 2010.
[19] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, Apr. 1975.
[20] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.