DePa: Simple, Provably Efficient, and Practical Order Maintenance for Task Parallelism

SAM WESTRICK, Carnegie Mellon University, USA
LARRY WANG, Carnegie Mellon University, USA
UMUT A. ACAR, Carnegie Mellon University, USA

A number of problems in parallel computing require reasoning about the dependency structure in parallel programs. For example, dynamic race detection relies on efficient “on-the-fly” determination of dependencies between sequential and parallel tasks (e.g. to quickly determine whether or not two memory accesses occur in parallel). Several solutions to this “parallel order maintenance” problem has been proposed, but they all have several drawbacks, including lack of provable bounds, high asymptotic or practical overheads, and poor support for parallel execution.

In this paper, we present a solution to the parallel order maintenance problem that is provably efficient, fully parallel, and practical. Our algorithm—called DePa—represents a computation as a graph and encodes vertices in the graph with two components: a dag-depth and a fork-path. In this encoding, each query requires \( O(f/\omega) \) work, where \( f \) is the minimum dynamic nesting depth of the two vertices compared, and \( \omega \) is the word-size.

In the common case (where \( f \) is small, e.g., less than 100), each query requires only a single memory lookup and a small constant number of bitwise instructions. Furthermore, graph maintenance at forks and joins requires only constant work, resulting in no asymptotic impact on overall work and span. DePa is therefore work-efficient and fully parallel.

Additional Key Words and Phrases: parallelism, sp-order maintenance, race detection

1 INTRODUCTION

With the mainstream availability of multicore computers, parallel programming today is important, relevant, and challenging. One of the chief factors that make it challenging is race conditions, which are difficult to avoid, debug, and fix. To help address this challenge, researchers have developed a variety of dynamic race detection algorithms which instrument a program to identify data races dynamically, i.e., while the program is running [8, 15, 17–19, 30, 36, 40–42, 45, 48, 49]. At a high level, these approaches broadly fall into two camps: those targeting general, coarse-grained concurrency [19, 36, 42, 49], and those targeting fine-grained, structured parallelism [8, 15, 17, 18, 30, 40, 41, 45, 48]. In this paper, we focus on the latter, specifically for nested fork-join parallelism (e.g. nested parallel-for loops), provided by systems such as Cilk [11, 21, 33], ParlayLib [9], OpenMP [38], Microsoft TPL [27], Intel TBB [24], Fork/Join Java [26], various forms of Parallel ML [6, 20, 22, 32, 37, 39, 46], and many others. Programs written in these languages and libraries typically consist of many (e.g., millions of) fine-grained tasks which are spawned and synchronized in a structured fashion.

At the heart of existing race-detection algorithms is an order maintenance data structure which encodes sequential dependencies between tasks [8, 15, 17, 18, 30, 40, 41, 45, 47, 48]. The purpose of this data structure is to enable efficient queries, to quickly determine whether or not two tasks are logically concurrent with one another.\(^1\) Race detection then can be performed by monitoring

\(^1\)Two tasks are logically concurrent if there exists a schedule in which their operations could be interleaved.
individual reads and writes: a data race is detected when two (logically) concurrent tasks both access the same memory location, and at least one of these tasks modifies the location.

In practice, the overheads of race detection are largely determined by the performance characteristics (and implementation details) of this order maintenance structure. As such, many solutions have been developed, specifically with the goals of low overhead both in theory and practice, and support for parallel execution. However, existing solutions come with a range of drawbacks. Some do not support parallel execution, or only provide limited support for it [8, 17, 40]. Others support parallel execution but incur either high theoretical overhead per query [30] or present no bound on this cost [41]. For nested fork-join programs, Utterback et al.’s algorithm [45] is asymptotically optimal; however, it is complex and difficult to implement, as it requires a custom scheduler which is tailored specifically to order maintenance. We therefore seek a solution which has provably low overhead, is fully parallel, and is simple to implement in practice.

Our solution is a “depths-and-paths” data structure, or DePa for short, which encodes the computation as a graph and performs precedence (reachability) queries between vertices. The DePa structure maintains only the dynamic task tree instead of the whole computation graph, and annotates each tree node with two quantities: (1) its depth in the graph, and (2) its path from the root of the task tree. We then prove that the depth and the path are sufficient for answering precedence queries on the computation graph. Essentially, the depth and path serve as two “coordinates” in a computation graph: together, these not only uniquely identifying a vertex, but also enable us to efficiently determine whether or not there is a path between two vertices.

For most computations the depth and path can fit into a few machine words and therefore the cost of precedence queries is a small constant. In particular, we prove (Theorem 3.6) that our algorithm requires only \( O(\min(f_u, f_v)/\omega) \) work for a query between two graph vertices \( u \) and \( v \). Here, \( f_u \) and \( f_v \) are (respectively) the dynamic nesting depths of \( u \) and \( v \), and \( \omega \) is the word size of the machine. This is effectively a constant bound in practice: with proper load-balancing and granularity control, dynamic nesting depths are typically small (e.g., less than 100). In terms of space, we show that the overhead of the algorithm is small: storing the depth and path of a vertex \( v \) requires only \( O(f_v/\omega) \) space, and any additional data required by the algorithm is bounded by \( O(PF + PF/\omega) \) where \( P \) is the number of processors and \( F \) is the maximum dynamic nesting depth of the computation.

In addition to supporting queries efficiently, our DePa algorithm is fully parallel. Maintaining the task tree requires only local computation with constant cost at each fork and join to update depths and paths. The work and span of a program therefore stays asymptotically unchanged by DePa maintenance, and hence so does its parallelism (the ratio of work to span). Queries are also parallel, as they can be implemented almost entirely with simple bitwise operations and arithmetic, and do not require any synchronization. Finally, to improve efficiency in practice, we present an optimization technique which integrates DePa with a work-stealing scheduler to improve data locality at queries. This optimization—although not essential—is easy to implement and requires only small changes to the scheduler itself.

In summary, the contributions of this paper include the following.

- The “depth-and-paths” (DePa) data structure for representing and querying task graphs for series-parallel ordering, along with a correctness proof (Section 3).
- Asymptotic analysis establishing work, span, and space bounds for DePa (Section 3.4).
- Optimization techniques for improving practical efficiency by integrating with work-stealing scheduling (Section 4).
2 PRELIMINARIES

Fork-join. We consider fork-join parallel programs which are based on tasks organized in a dynamic task tree. Initially, there is a single “root” task. At any moment, any leaf (a task with no children) may fork, which creates two child tasks. While the children are executing, the parent task is suspended. As soon as both children have completed, they join with the parent, which deletes the children from the tree and resumes execution of the parent task. At any point of program execution, the tasks at leaf nodes of the task tree are said to be running while the tasks at internal nodes are said to be suspended. Any task which is either suspended or running is said to be active. We say that two tasks are concurrent if neither is an ancestor of the other. That is, in the task tree, concurrent tasks could be siblings, cousins, etc.

Dags (Computation Graphs). A fork-join computation can be summarized with a directed, acyclic graph called a dag [4]. The dag represents the history of a completed computation: each vertex represents a sequence of executed instructions uninterrupted by a fork or join, and edges are ordering constraints. We can construct a dag dynamically during execution by (1) creating new outgoing edges to two new vertices at each fork, and (2) creating two incoming edges to a single new vertex at each join. We say that vertex \( u \) precedes vertex \( v \), denoted \( u \preceq v \), if there exists a path in the dag from \( u \) to \( v \). Note that \( \preceq \) is a partial order.

Series-Parallel Dags. The computation graphs of (nested) fork-join parallel programs are series-parallel graphs, i.e. inductively constructed by sequential and parallel composition. This is illustrated in Figure 1.

3 DEPA ALGORITHM

We present an order maintenance data structure which efficiently supports queries of the form \( u \preceq_G v \), i.e., determining whether or not there is a path from \( u \) to \( v \) in computation graph \( G \). At a high level, our algorithm assigns each vertex a unique vertex identifier consisting of two components: a dag-depth and a fork-path. Together, the dag-depths and fork-paths of vertices allow us to identify vertices and efficiently determine their relative positions in the dag. We call our algorithm DePa for its combined use of depths and paths.

3.1 Dag-Depths and Fork-Paths

The dag-depth of a vertex is the length of the longest path that ends at that vertex. The fork-path of a vertex encodes the nesting of its corresponding task as a sequence of bits. Each bit is one of \( L \) or \( R \), to respectively indicate a fork either on the left or the right. We denote dag-depths with \( D(v) \) and fork-paths with \( P(v) \). Furthermore, we denote the longest common prefix between two fork-paths with \( \text{LCP}(p_1, p_2) \).

Dag-depths and fork-paths can be computed solely in terms of a vertex’s immediate predecessors, as shown in Figure 2. When a vertex \( u \) forks into \( v \) and \( w \), the new vertices have depth \( D(v) = D(w) = 1 + D(u) \), and their paths are extended with \( L \) and \( R \); that is, \( P(v) = P(u) \oplus L \) and \( P(w) = P(u) \oplus R \) (where \( \oplus \) denotes concatenation). When two vertices \( u \) and \( v \) join, they must be siblings, i.e. their paths must be \( p \oplus L \) and \( p \oplus R \) for some common prefix \( p \). The new vertex then has path \( p \) and depth \( 1 + \max(D(u), D(v)) \). An example of a full dag is shown in Figure 3, where we write “—” for the empty path.

Intuitively, it’s helpful to think of the dag-depth and fork-path as two “coordinates” into a dag. When illustrating a dag with edges pointing down, the dag-depth serves as a vertical position, and the fork-path is a horizontal position. Together, these uniquely identify a vertex.\(^2\)

\(^2\)We leave this to intuition, and prove a more general result in Lemma 3.1.
3.2 Identifying Precedence

When two vertices have the same fork-path, one must precede the other in the dag. More generally, whenever one vertex’s fork-path is a prefix of another, one of the two vertices must precede the other. This precedence can be determined by comparing their dag-depths, as stated in Lemma 3.1.

**Lemma 3.1.** Let \( u \) and \( v \) be any two vertices in dag \( G \) such that \( P(u) \) is a prefix of \( P(v) \). Then (1) \( u \preceq_G v \) if and only if \( D(u) \leq D(v) \), and (2) \( v \preceq_G u \) if and only if \( D(v) \leq D(u) \).

**Proof.** We will only present a proof of (1), as (2) is symmetric. The forward direction is straightforward: if \( u \preceq_G v \), then there exists a path in the dag from \( u \) to \( v \), so \( D(u) \leq D(v) \). To prove the reverse direction, we induct over the structure of \( G \). Consider \( u \) and \( v \) such that \( P(u) \) is a prefix of \( P(v) \) and \( D(u) \leq D(v) \). There are three cases for the structure of \( G \):

- If \( G \) is a single vertex, the lemma holds trivially.
- Suppose \( G \) is the serial composition of \( G_1 \) and \( G_2 \). If \( u \) and \( v \) either both occur in \( G_1 \) or both occur in \( G_2 \), then \( u \preceq_G v \) by induction. Otherwise, due to \( D(u) \leq D(v) \), we must have that \( u \) occurs in \( G_1 \) and \( v \) occurs in \( G_2 \). Consider the vertex \( r \) where the two sub-dags overlap. This vertex \( r \) is the sink of \( G_1 \), so \( u \preceq_G r \). Similarly, \( r \) is the source of \( G_2 \), so \( r \preceq_G v \). Together, these imply \( u \preceq_G v \).
- Suppose \( G \) is the parallel composition of \( G_1 \) and \( G_2 \). If \( u \) is the source vertex of \( G \) or if \( v \) is the sink vertex of \( G \), then clearly we have \( u \preceq_G v \). If \( u \) and \( v \) either both occur in \( G_1 \) or both occur in \( G_2 \), then \( u \preceq_G v \) by induction. Otherwise, suppose WLOG that \( u \) occurs in \( G_1 \) and \( v \) occurs in \( G_2 \). This violates the assumption that \( P(u) \) is a prefix of \( P(v) \), because all fork-paths of vertices in \( G_1 \) begin with \( p \oplus R \) and all in \( G_2 \) begin with \( p \oplus L \) (where \( p \) is the fork-path of the source of \( G \)).

□

A direct application of this lemma reveals that our vertex identifiers do indeed uniquely identify vertices.

**Corollary 1.** For any two vertices \( u \) and \( v \), \( u = v \) if and only if \( P(u) = P(v) \) and \( D(u) = D(v) \).

We can also obtain a method for checking precedence between any two arbitrary vertices. To do so, we identify a special **critical** vertex, denoted \( C(u, v) \), defined below. Note that the critical vertex is always well-defined for any pair of vertices, because for every prefix of a vertex’s fork-path there exists a vertex with less-or-equal dag-depth.
DePa: Simple, Provably Efficient, and Practical Order Maintenance for Task Parallelism

**Definition 1** (Critical Vertex). Consider any two vertices $u$ and $v$, and WLOG assume $D(u) \leq D(v)$. The critical vertex $C(u, v)$ is defined as the vertex $c$ such that $P(c) = LCP(P(u), P(v))$ and $D(c)$ is maximized s.t. $D(c) \leq D(v)$.

As formalized below in Lemma 3.2, the critical vertex is special in that if a path $u \preceq v$ exists, then $C(u, v)$ will certainly lie on the path. Additionally, if there is not a path between the two vertices, then the critical vertex serves as proof that the path does not exist.

**Lemma 3.2.** Let $u$ and $v$ be any two vertices, and let $c = C(u, v)$ be their critical vertex. Then $u \preceq v$ if and only if $D(u) \leq D(c) \leq D(v)$.

**Proof.** We separately consider the “if” and “only-if” directions. For the “if” direction, suppose $D(u) \leq D(c) \leq D(v)$. Then by Lemma 3.1, we have both $u \preceq c$ and $c \preceq v$, and therefore $u \preceq v$. Note that the Lemma is applicable because $P(c)$ is a path according to $D(c) \leq D(v)$. For the “only-if” direction, suppose $u \preceq v$, i.e. that there exists a path from $u$ to $v$ in the dag. Consider the sequence of vertices in order along the path from $u$ to $v$. For each adjacent pair of vertices, their fork-paths differ only by either pushing or popping one element at the end. Therefore, at least one vertex along the path has fork-path $LCP(P(u), P(v))$. The last such vertex along the path is $c$, as it has maximum dag-depth. Because $c$ is on the path, and because dag-depths strictly increase along each path, we have $D(u) \leq D(c) \leq D(v)$. \(\square\)

**Relationship with Task Trees.** Observe that whenever a leaf task suspends (i.e. when it forks), at that moment its current vertex has larger dag-depth than any task which has ever had the same fork-path. The vertices of suspended ancestors are therefore critical vertices for their children in the task tree, as these vertices cover all possible prefixes of the fork-path and have maximum dag-depths for those fork-paths. In particular, for queries of the form $u \preceq v$ where $v$ is the current vertex of a leaf task (which are exactly the sort of queries needed for entanglement detection), the critical vertex $C(u, v)$ will be the current vertex of one of the task’s suspended ancestors.

### 3.3 Algorithm

The DePa order maintenance algorithm is presented in Figure 4. It consists of three components: forks, joins, and precedence queries. As described in Lemma 3.2, precedence queries rely on knowing the depths of critical vertices. Accordingly, the algorithm maintains a global lookup data structure called SuspendedDepths which maps fork-paths of suspended ancestors to their dag-depths. (In theory, this could be implemented simply as a hash table; we provide a detailed discussion of an optimized implementation in practice to Section 4.)

At each fork, the algorithm initializes new vertex identifiers for the two new tasks, and updates the SuspendedDepths data structure to remember the dag-depth of the task that forked (which is now suspended). At each join, in preparation of resuming the parent task, the algorithm constructs a new vertex identifier for the parent and removes the old value from the SuspendedDepths data structure. At each query, the algorithm computes the longest-common-prefix of the vertices’ fork paths and looks up the appropriate dag-depth to infer precedence.

**Theorem 3.3 (DePa Correctness).** Let $v$ be the current vertex of some leaf task. Then for any other vertex $u$, the order maintenance algorithm will correctly report that $u$ precedes $v$ if and only if $u \preceq v$.

**Proof.** Let $p = LCP(P(u), P(v))$. There are two cases to consider: when $p = P(v)$ and when $p \neq P(v)$. If $p = P(v)$, then $C(u, v) = v$. In this case, the algorithm reports (Figure 4, line 23) that there is a path according to $D(u) \leq D(v)$. By Lemma 3.2, since $C(u, v) = v$, this is correct. Next, suppose $p \neq P(v)$. Then there is a suspended ancestor whose fork-path is $p$; let $c$ be the vertex of this ancestor,
and note that SuspendedDepths[p] stores D(c). We know that c = C(u,v) and also that D(c) ≤ D(v)
because v occurs in a descendant task. Therefore, by Lemma 3.2, the algorithm correctly reports
(Figure 4, line 24) that there is a path according to D(u) ≤ SuspendedDepths[p]. □

3.4 Cost Analysis

**Lemma 3.4 (Vertex Identifier Space).** Each vertex identifier can be compactly represented using
O(f/ω) words, where f is the dynamic nesting depth of the corresponding task, and ω is the word-size.

**Proof.** The dag-depth component requires one word, and the fork-path is a bit sequence of
length f. Together, these require 1 + [f/ω] words. □

**Theorem 3.5 (DePa Space).** At any point in a P-processor execution, the algorithm requires
O(PF + PF/ω) additional space, where F is the maximum dynamic nesting depth of the computation.

**Proof.** There are two sources of space overhead: (1) each running task stores its current ver-
tex identifier, and (2) the SuspendedDepths data structure. We assume here that the scheduler
guarantees at most O(P) running tasks at any moment (typical schedulers such as work-stealing
have this guarantee). Each of these stores a vertex identifier, requiring O(PF/ω) space. In the
SuspendedDepths data structure, each running task has at most F suspended ancestors, therefore
the size of SuspendedDepths at any moment is at most O(PF). □

**Theorem 3.6 (DePa Query Cost).** A query between vertices u and v costs O(min(f_u, f_v)/ω) work,
where f_u and f_v are the dynamic nesting depths of u and v, respectively.

**Proof.** Using the representation of Lemma 3.4, computing the longest common prefix between
fork-paths requires O(min(f_u, f_v)/ω) work. We can perform the SuspendedDepths lookup within
the same bound by representing it as a hash table. The rest of the operations require constant
work. □

**Theorem 3.7 (DePa Fork and Join Cost).** DePa requires O(1) work and span at each fork and
join to update depths and paths.

**Proof.** By representing paths as a linked list of words, we can update paths at forks and joins
with O(1) work. The SuspendedDepths data structure can be implemented as a hash table with
paths as keys. Each vertex can additionally store its own hash, so that insertions and deletions (at forks and joins) are $O(1)$. □

4 INTEGRATION WITH WORK-STEALING

For SP-order maintenance to be fast in practice, it is important not only to optimize individual checks, but also to make sure that forks and joins are fast. Much of the work of our order-maintenance algorithm at forks and joins is “local” in the sense that it only requires accessing the data of a task and its children, which should be fast in practice and parallelize well. There is one operation, however, which could potentially incur a significant overhead: updating the SuspendedDepths lookup data structure of the DePa algorithm (Figure 4).

In the cost theorem (Theorem 3.7), we assume that the SuspendedDepths data-structure provides $O(1)$ updates and deletions, with no contention. In practice, a straightforward implementation such as a global shared hash table may incur significant contention on concurrent accesses. In this section, we describe how to improve efficiency by integrating the SuspendedDepths data structure with a work-stealing scheduler. The basic idea is avoid contention with a “copy-on-steal” approach, where each processor locally stores the portion of the data-structure it needs.

Implementing SuspendedDepths. Throughout execution, each processor’s current task is a leaf in the task tree. A key observation is that, for our order-maintenance algorithm, each processor only needs access to the dag-depths of tasks along its root-to-leaf path in the tree. Therefore, we can implement SuspendedDepths in a distributed fashion, using one array stored locally on each processor, where each array (of length up to $F$, the maximum nesting depth) stores a root-to-leaf path of suspended depths.

Each processor-local array operates like a stack. Immediately before each fork, the current processor pushes the current dag-depth onto its local array. Immediately after each join, the current processor pops one element. To guarantee that processors locally have the suspended depths of all ancestor tasks, we copy a small amount of data on each steal: when a processor steals a task that has $K$ ancestors, it copies $K$ suspended depths from the victim into its own local array. This requires $O(F)$ work on each steal, which in practice is effectively free, because $F$ is typically small, and the cost of steals is often well-amortized with appropriate granularity control.

5 RELATED WORK

In 1974, Brent proved that a parallel program with total work $W$ and span (depth) $S$ can be executed on $P$-processor, in $T_P \leq W/P + S$ time. This seminal result has underpinned decades of research on scheduling algorithms for task parallelism. This research have shown that this bound is not purely theoretical and can in fact be realized efficiently in practice [1–3, 7, 12, 14, 16, 21, 23, 35], usually by using a variant of work stealing. Recent work has also made some progress extending this scheduling theory to account for latency [31] and competition among tasks, by allowing tasks to declare priorities [34, 43].

The advances on scheduling algorithms it turn lead to the development of many task-parallel programming languages and systems in both functional and procedural languages.

An important concern is all these task-parallel systems is race detection. Because data races usually cause incorrect behavior, especially at scale [5, 13], there has been much work on detecting races in task parallel, such as fork-join programs, have been proposed [8, 15, 17, 18, 30, 40, 41, 45, 48]. These algorithms all revolve around an ordering data structure, that allows determining whether two instructions are sequentially dependent or can be executed in parallel. The basic idea is to check at each memory access whether the instruction performing the access creates a race with the instructions that accessed the same location in the past by using the ordering data structure.
The theoretical and practical efficiency of these algorithms therefore critically depend on the series-parallel order data structure. Of the prior approaches, some do not guarantee constant work/time bounds [29, 30, 40, 41]. Others run only sequentially [15, 17] or limit parallelism with non-constant overheads [8]. More recent work can guarantee constant overheads by carefully integrating the ordering data structure and the scheduler but this comes with significant complexity in managing concurrency [45]. All of the above work considers nested parallelism with fork-join and async-finish constructs, which result in similar dependency structures. More recent work considers race detection for futures and establishes worst-case bounds, though the overheads are no longer constant [48].

In comparison to this related work, our DePa algorithm is perhaps most similar to Mellor-Crummeys offset-span vertex labeling technique [30]. An offset-span labeling for a vertex \( v \) requires \( O(f_0) \) space, where \( f_0 \) is the dynamic nesting depth of that vertex; similarly, the worst-case time required for a query is \( O(F) \) where \( F \) is the maximum dynamic nesting depth. These bounds may at first appear essentially identical to those achieved by DePa, if one ignores the factor \( \omega \) (the word-size) in our bounds. However, we specifically included this factor in the bounds as it contributes to a significant savings in practice. For any vertex \( v \), the offset-span label of \( v \) requires approximately \( 2f_0 \) words; in contrast, the DePa label for the same vertex requires only \( 1 + \lceil f_0/\omega \rceil \) words. For typical word sizes (e.g. \( \omega = 64 \)), this is between one and two orders of magnitude improvement in space. The cost of queries is likely also reduced, although a direct comparison will be needed to determine the exact difference in overhead.

Race detection has also been studied extensively in the more general concurrency setting, where programs contain a small number of coarse-grained threads that may synchronize by using locks, synchronization variables, etc. Techniques from this second line of work do not scale to task-parallel programs, because of their coarse-grained threads assumption (e.g., [41]), and is less directly relevant to this paper.

Early work on coarse-grained threads proposes the lock-set algorithm [42], which can lead to false positives. Subsequent work proposed precise techniques by using vector-clocks to capture the happens-before relations between threads [19]. Followup work has proposed hybrid approaches that combine lock sets and vector clocks, trading off efficiency and precision [36, 49]. Most approaches use dynamic, on-the-fly race detection though there has also been some work on predicting data races [25, 44].

Dynamic race-detection techniques in coarse-grained multithreaded programs are typically sensitive to scheduling decisions: because they track actual threads, the techniques detect races in the run being checked and remain sensitive to scheduling decisions. In contrast, prior race detection techniques for task parallel programs are able to account for logical (potentially unrealized) parallelism [8, 15, 17, 29, 30, 40, 41, 45].

**Pedigrees and DPRNGs.** Any scheme which uniquely identifies individual “vertices” of a computation graph while ignoring details of scheduling is known as a **pedigree** scheme [28]. Pedigrees are potentially useful for a variety of applications. One such application is a deterministic parallel random-number generator (or **DPRNG**, for short), where pedigrees can be used as seeds for the generator. For example, Leiserson et al. [28] describe a particular pedigree scheme and use these pedigrees to implement a DPRNG with low overhead in practice. DPRNGs make it possible to write parallel randomized algorithms with repeatable behavior: because pedigrees are independent of scheduler decisions, this approach is deterministic under parallel execution. This kind of repeatable behavior is a form of **internal determinism**, which provides a number of benefits, such as simplifying the process of debugging and performance tuning [10].
Because DePa provides a unique labeling of vertices within a computation graph, it could be used as an efficient pedigree scheme for a DPRNG. In comparison to the pedigree scheme of Leiserson et al. ([28]), one potential advantage of DePa is in reducing the size of pedigrees; specifically, DePa labels are approximately a factor $\omega$ smaller, where $\omega$ is the word-size of the machine.

6 CONCLUSION

We present DePa, a novel solution to the so-called SP maintenance problem which is key to the efficiency of techniques such as dynamic race detection. The gist of our approach is to represent a computation as a graph, where each vertex is labeled with both its depth in the graph as well as the fork path of its corresponding nested task. This encoding is compact and enables fast precedence queries between vertices. DePa therefore provides a number of advantages in comparison to prior techniques: (1) it is provably efficient with low overhead in terms of both space and time, (2) it is fully parallel, and (3) it is simple to implement.
REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. Theory of Computing Systems, 35(3):321–347, 2002.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13, 2013.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. Journal of Functional Programming (JFP), 26:e23, 2016.

[4] Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. Dag-calculus: A calculus for parallel computation. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pages 18–32, 2016.

[5] Sarita V. Adve. Data races are evil with no exceptions: technical perspective. Commun. ACM, 53(11):84, 2010.

[6] Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space efficient parallel functional programming. In Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL), 2021.

[7] Nimir S. Arora, Robert D. Blumofe, and Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems, 34(2):115–144, 2001.

[8] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In 16th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 133–144, 2004.

[9] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib - A toolkit for parallel algorithms on shared-memory multicore machines. In Christian Scheideler and Michael Spear, editors, SPAA ’20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020, pages 507–509. ACM, 2020.

[10] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In PPoPP ’12, pages 181–192, 2012.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing, 37(1):55 – 69, 1996.

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. Proc. ACM 46:720–748, September 1999.

[13] Hans-Juergen Boehm. How to miscompile programs with “benign” data races. In 3rd USENIX Workshop on Hot Topics in Parallelism, HotPar’11, Berkeley, CA, USA, May 26-27, 2011, 2011.

[14] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In Functional Programming Languages and Computer Architecture (FPCA ’81), pages 187–194. ACM Press, October 1981.

[15] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures, SPAA ’98, 1998.

[16] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling - A status report. In Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004, Revised Selected Papers, pages 1–16, 2004.

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

[18] Jeremy T. Fineman. Provably good race detection that runs in parallel. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018, pages 81–93, 2018.

[19] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. SIGPLAN Not., 44(6):121–133, June 2009.

[20] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. Journal of Functional Programming, 20(5-6):1–40, 2011.

[21] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In PLDI, pages 212–223, 1998.

[22] Adrien Guatto, Sam Westrick, Ram Raghuamnathan, Umut A. Acar, and Matthew Fluet. Hierarchical memory management for mutable state. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018, pages 81–93, 2018.

[23] Robert H. Halstead. Multisp: a language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems, 7:501–538, 1985.

[24] Intel. Intel threading building blocks, 2011. https://www.threadingbuildingblocks.org/.

[25] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In Albert Cohen and Martin T. Vechev, editors, Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, pages 157–170. ACM, 2017.
DePa: Simple, Provably Efficient, and Practical Order Maintenance for Task Parallelism

[26] Doug Lea. A Java fork/join framework. In Proceedings of the ACM 2000 conference on Java Grande, JAVA ’00, pages 36–43, 2000.

[27] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications, OOPSLA ’09, pages 227–242, 2009.

[28] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’12, 2012.

[29] Li Lu, Weixing Ji, and Michael L. Scott. Dynamic enforcement of determinism in a parallel scripting language. In Michael F. P. O’Boyle and Keshav Pingali, editors, ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014, pages 519–529. ACM, 2014.

[30] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In Proceedings of Supercomputing ’91, pages 24–33, 1991.

[31] Stefan K. Muller and Umut A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016, pages 71–82, 2016.

[32] Stefan K. Muller, Umut A. Acar, and Robert Harper. Responsive parallel computation: Bridging competitive and cooperative threading. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pages 677–692, New York, NY, USA, 2017. ACM.

[33] Stefan K. Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. Responsive parallelism with futures and state. In Alastair F. Donaldson and Emina Torlak, editors, Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020, pages 577–591.

[34] Stefan K. Muller, Sam Westrick, and Umut A. Acar. Fairness in responsive parallelism. In Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming, ICFP 2019, 2019.

[35] Girija J. Narlikar and Guy E. Blelloch. Space-efficient scheduling of nested parallelism. ACM Transactions on Programming Languages and Systems, 21, 1999.

[36] Robert O’Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In Rudolf Eigenmann and Martin C. Rinard, editors, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2003, June 11-13, 2003, San Diego, CA, USA, pages 167–178. ACM, 2003.

[37] Atsushi Ohori, Kenjiro Taura, and Katsuhiro Ueno. Making sml# a general-purpose high-performance language, 2018. Unpublished Manuscript.

[38] OpenMP Application Programming Interface, Version 5.0. November 2018. Accessed in July 2018.

[39] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical memory management for parallel programs. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pages 392–406, New York, NY, USA, 2016. ACM.

[40] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, Runtime Verification, volume 6418 of Lecture Notes in Computer Science, pages 368–383. Springer Berlin / Heidelberg, 2010.

[41] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise datarace detection for structured parallelism. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12, pages 531–542, 2012.

[42] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP), October 1997.

[43] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. Priority scheduling for interactive applications. In Christian Scheideler and Michael Spear, editors, SPAA ’20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020, pages 465–477.

[44] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In John Field and Michael Hicks, editors, Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012, pages 387–400. ACM, 2012.

[45] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 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, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016, pages 83–94, 2016.
[46] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. In Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL), 2020.

[47] Yifan Xu, Kunal Agrawal, and I-Ting Angelina Lee. Efficient parallel determinacy race detection for structured futures. In Kunal Agrawal and Yossi Azar, editors, SPAA ’21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021, pages 398–409. ACM, 2021.

[48] Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. Parallel determinacy race detection for futures. In Rajiv Gupta and Xipeng Shen, editors, PPoPP ’20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020, pages 217–231. ACM, 2020.

[49] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In Andrew Herbert and Kenneth P. Birman, editors, Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005, pages 221–234. ACM, 2005.