Efficient Parallel Determinacy Race Detection for Structured Futures

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

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

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

Abstract

In task-parallel code, a ***determinacy race*** occurs when two *logically* parallel instructions access the same memory location in a conflicting way. A determinacy race tends to be a bug as it leads to non-deterministic program behaviors.

Researchers have studied algorithms for detecting determinacy races in task-parallel code, with most prior work focuses on computations with nice structural properties (e.g., fork-join or pipeline parallelism). For such computations, one can devise provably efficient algorithms with constant overhead, leading to a asymptotically optimal running time — $O(T_1/P + T_\infty)$ on P cores for a computation with T_1 work and T_∞ span.

More recently, researchers have begun to address the problem of race detecting computations with less structural properties, such as ones that arise from the use of futures. Due to the lack of structural properties, the race detection algorithm incurs higher overhead. Given a computation with work T_1 and span T_∞, the state-of-the-art parallel algorithm for race detecting programs with futures runs in time $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(\lg k + \lg r \lg \hat{k}))$ on P cores (Xu et al., 2020), where k is the total number of futures used, $\hat{k}$ is the maximum number of future operations per "future task," and r is the maximum number of readers between two consecutive writes to a given memory location.

Interestingly, it has been shown that when one imposes certain restrictions on the use of futures, referred to as the ***structured futures***, although the restrictions do not entirely eliminate arbitrary dependences among subcomputations, one can race detect such programs more efficiently than that for programs with general futures (i.e., no restrictions). The improved efficiency has only been demonstrated for a sequential algorithm (Utterback et al., 2019) that race detects while executing the computation *sequentially*, however. The algorithm requires sequential execution, because the correctness of the algorithm relies on updating the necessary data structures while traversing the computation dag in a specific order. An interesting question remains, whether a parallel algorithm exists for race detecting programs with structured future that achieves better execution time compared to that designed for general futures.

This work attempts to answer this question. We propose a parallel algorithm designed to race detect structured futures. By exploiting the restrictions imposed by structured futures, the proposed algorithm allows for a constant-time query while keeping fewer previous accessors around to provide the same correctness guarantees as prior work. Our algorithm runs in time $O((T_1 + k^2)/P + T_\infty \lg k)$ on P cores.

We have also implemented and empirically evaluated the proposed algorithm. When compared to the state-of-the-art sequential algorithm designed for structured futures, although our algorithm has a longer one-core execution time, its absolute running time wins out when running on two cores or more. When compared to the state-of-the-art parallel algorithm designed for general futures, it indeed incurs lower overhead and performs better.

CCS Concepts

• Theory of computation → Dynamic graph algorithms; **• Software and its engineering → Software testing and debugging**.

Keywords

determinacy race; race detection; reachability analysis; future

ACM Reference Format:
Yifan Xu, Kunal Agrawal, and I-Ting Angelina Lee. 2021. Efficient Parallel Determinacy Race Detection for Structured Futures. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3409964.3461815

1 Introduction

Task parallelism is designed to simplify the job of programming multicore hardware — in this paradigm, the programmer expresses the *logical* parallelism of the computation using high-level language constructs. The underlying runtime scheduler is responsible for automatically scheduling and load balancing the computation on the parallel machine. Examples of task parallel platforms include OpenMP [30], Intel's TBB [21, 33], IBM's X10 [8], various Cilk dialects [11, 17, 22–24], and Habanero dialects [3, 7].

Even though task parallelism simplifies the parallel programming, a common programming pitfall, called a ***determinacy race*** [13],[1] remains. A determinacy race occurs when two *logically* parallel instructions access the same memory location in a conflicting way (i.e., at least one is a write). In the absence of determinacy race, task-parallel computations behave deterministically. On the other hand, determinacy races can lead to nondeterministic program behaviors and therefore they tend to be bugs. They are

[1]The literature sometimes refers to a determinacy race as a ***general race*** [28].

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 ACM 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 '21, July 6–8, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8070-6/21/07…$15.00
https://doi.org/10.1145/3409964.3461815

often challenging to detect and diagnose because they may not manifest in every execution.

To address this challenge, researchers have designed algorithms for automatic detection of determinacy races in task-parallel code [1, 4, 12–15, 26, 31, 32, 37, 39, 40, 42, 43]. This work focuses on ***on-the-fly race detection***, where the algorithm checks for races[2] while the program executes on a particular input. These systems provide strong correctness guarantees: the detector reports a race if and only if the program has a race on that input.

Typically, an on-the-fly race detector consists of two main components. The first component, the ***reachability analysis***, answers the query of whether two instructions are logically in parallel. The second component, the ***access history***, remembers what instructions accessed a given memory location previously. During execution, whenever an instruction v accesses a memory location l, the detector checks with the access history to find any prior conflicting accesses, say u, and queries the reachability component to see if u and v are logically in parallel; if so, a race is reported.

Much of the prior work on race detection for task-parallel programs has focused on structured parallelism such ***fork-join*** parallelism [4, 13, 15, 26, 29, 39] and pipeline parallelism [12, 42]. These programs have structural properties that make both the reachability analysis and access history efficient. In particular, it turns out that two total orders are sufficient to perform reachability analysis [29, 42], and the access history needs to contain only a constant number of prior accessors per memory location [26, 42]. By exploiting the structural properties, the state-of-the-art race detection algorithms for fork-join [39] and pipeline [42] parallelism can race detect while executing the computation in parallel with constant overhead. That is, given a computation with T_1 ***work*** — its running time on one core — and T_∞ ***span*** — the longest dependences in the computation, or its running time on infinitely many cores — these parallel algorithms run in time $O(T_1/P + T_\infty)$ on P cores, which is the best one can hope for.

There has also been recent work on race detection for less structured programs [1, 37, 40, 43], such as those that contain ***futures*** [18]. Futures allow for arbitrary dependences making race detection more expensive. Most of these race detection algorithms [1, 37, 40] are *sequential*; that is, the algorithm race detects while execute the computation *sequentially*. The requirement of a sequential execution is fundamental — the correctness of the reachability analysis in these algorithms relies on traversing the computation dag in a particular sequential (i.e., left-to-right depth-first) traversal order. Furthermore, storing a constant number of accessors no longer suffices for the access history. Instead, one may store as many as r accessors per memory location to not miss a race, where r is the maximum number of readers between a pair of writes.

To the best of our knowledge, the only prior result that performs race detection on programs with futures in parallel is by Xu et al. [43]. Due to the lack of structural properties, the reachability component in this parallel algorithm incurs $O(k^2)$ overhead for construction and $O(\lg \hat{k})$ for each query, where k is the total number of futures used in the computation,[3] and $\hat{k}$ is the maximum number of future operations within a single "future task" (formally defined in Section 2). They must still store r accessors, leading to the overall running time of $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(\lg k + \lg r \lg \hat{k}))$ for a program with work T_1 and span T_∞ running on P cores.[4]

Interestingly, the work by Utterback et al. [40] explores a sequential algorithm for race detecting programs with ***structured futures***, which imposes certain restrictions on how futures can be used (more details in Section 2). Even though structured futures still allows for arbitrary dependences among subcomputations, these programs still have more structural properties compared to an ***general*** futures (i.e., no restrictions) that allow for more efficient race detection. Utterback et al. gave a sequential algorithm for reachability analysis with an *almost* constant amortized overhead giving a total running time of approximately $O(T_1)$. However, this algorithm is inherently sequential and heavily depends on the depth-first left-to-right execution of the program.

Beyond the race detection work by Utterback et al. [40], it has also been shown that the structured futures allow one to achieve better bounds on cache misses [19] and scheduling overhead [35] compared to that for general futures. Such results are interesting because the set of programs generated by structured futures is larger than the set generated by fork-join and pipeline parallelism and contains them both. Moreover, the use of structured futures is not purely of academic interests but useful in practice. The scheduling work by Singer et al. [35] showed that one can implement dynamic programming applications such as Smith-Waterman sequence alignment with lower span compared to the implementation with only fork-join parallelism (albeit the improvement is constant and not asymptotic) and thereby achieve better scalability in practice. Other platforms that employ futures (e.g. [27, 34]) were also able to utilize structured futures to implement interesting application features that traditional fork-join parallelism could not achieve.

In this paper, we propose a parallel race detection algorithm for programs with structured futures. By exploiting the restrictions imposed by the structured use of futures, we are able to bring down the reachability query overhead to be constant time (although the construction overhead is still $O(k^2)$), and we are able to bound the number of readers to keep per memory location. Specifically, one can retain the same correctness guarantees while storing at most $2k$ readers per memory location. Combining these savings in overhead, our algorithm runs in time $O((T_1 + k^2)/P + T_\infty \lg k)$ on P cores, where k is the total number of futures used in the computation. The interesting thing to note is that, unlike the prior results for race detecting general futures [43], this bound does not depend on r, the number of readers between a pair of writes. In addition, compared to the bound by prior work, this running time provides a saving of a $\lg \hat{k}$ multiplicative factor on the work term and a $\lg r \lg \hat{k}$ additive factor on the overhead on the span term.

[2]Henceforth, we refer to a determinacy race as simply a race.

[3]In the work by Xu et al. [43], k is used to refer to the total number of future operations. However, since their work assume constant number of future operations per future used, the bound remains the same even though we have changed what the term means.

[4]The original bound stated in Xu et al.'s paper was $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(k + \lg r \lg \hat{k}))$. However, it turns out that one can tighten the bound slightly by applying additional parallelism when maintaining the reachability data structured. Here, we state the improved bound assuming this additional parallelism.

We have implemented this algorithm in practice and empirically compared it against the state-of-the-art algorithms [40, 43]. Empirical results indicate that, when compared with the sequential algorithm designed for structured futures, although our algorithm has a slightly higher overhead, its absolute running time wins out when running on two cores or more. When compared to the parallel algorithm designed for general futures, our algorithm indeed incurs lower overhead and performs better.

2 Preliminaries

Fork-Join and Future Parallelism

This work assumes that the target program can be written with language constructs to express fork-join and future parallelism.[5] Most task-parallel platforms support fork-join parallelism (e.g., [3, 7, 8, 11, 17, 21–24, 30, 33]). Future parallelism, also supported in many platforms (e.g., [7, 8, 16, 18, 20, 25, 35, 38]), allows one to express parallelism in a more flexible manner compared to fork-join parallelism. We discuss how the language constructs work in Cilk dialects [17, 35]; other platforms may support them with different syntactic form, but the concepts are similar.

Fork-join parallelism in our model is expressed using two keywords: `spawn` and `sync`. A function F can ***spawn*** another function G by prefixing the invocation with the `spawn` keyword, which indicates the call to G may execute in parallel with the continuation of F after the `spawn` statement. The `sync` keyword serves as its counterpart, indicating the the control cannot pass the `sync` statement until all previously spawned subroutines have returned.

Future parallelism can be expressed using two keywords, `create` and `get`. The keyword `create` works similarly as `spawn`— when a function F invokes another function G with `create`, the invocation to G may execute in parallel with the continuation of F. The key distinction between `create` and `spawn` is that, `create` returns a ***handle*** — an object in memory that represents the execution of the created ***future task*** G — that can later be used to ensure the completion of G and query its result. The keyword `sync` has no effect on subroutines invoked with `create`. Rather, the termination of G is guaranteed by invoking a `get` on the handle associated with G; the control cannot pass `get` until G finishes with `get` returning its return value. Since the future handle can be stored in memory and retrieved at a later program point, futures allow for more flexible expression of parallelism.

The Computation Dag Model

The execution of a task-parallel program for a given input can be modeled as a ***directed acyclic graph*** (or ***dag*** for short), where a node represents a sequence of instructions containing no parallel control constructs, and an edge represents a dependence. We say that a node is ***ready*** to execute when all its immediate predecessors have executed.

During execution, the `spawn` keyword creates a ***spawn node*** with two outgoing edges, a ***spawn edge*** leading to the first node is the spawned subroutine and a ***continuation edge*** leading to the continuation of the caller. The `sync` keyword creates a ***sync node*** with multiple incoming edges, one from each spawned subroutine that the `sync` keyword is joining, and the sync node itself represents the continuation after the `sync` statement.

When only `spawn` and `sync` keywords are used, the execution forms a ***series-parallel dag*** (or ***SP-dag*** for short) [41], with the following properties. An SP-dag has a single ***source*** node which precedes all other nodes in the corresponding SP-dag and single ***sink*** node that comes after all nodes in the dag. Moreover, an SP-dag can be constructed recursively using series and parallel compositions. In a series composition of two SP-dags, we add an edge from the sink of one SP-dag to the source of the other, creating a new SP-dag. In the parallel composition, we create a new source with edges to both sources and a new sink with edges from both sinks. [6]

Adding futures can create arbitrary dependences among subcomputations. The execution of a `create` generates a ***create node*** with two outgoing edges — a ***create edge*** leading to the first node of the created future task and a continuation edge leading to the continuation of the caller. The execution of a function instance representing a future task terminates with a ***put node***, which is the last node to execute in the future task that deposits result into the associated future handle. Finally, the execution of `get` terminates the current node u and generates a ***get node*** with two incoming edges — an ordinary edge from u and a ***get edge*** from the put node of the corresponding gotten future task.

One can model the execution of a program with both fork-join and future parallelism as a set of SP-dags with arbitrary dependences between them. Each future task, an instance of a function execution possibly containing `spawn` and `sync` keywords, can be modeled as its own SP-dag, and the create and get edges form arbitrary dependences among them. Henceforth, we refer to the create and get edges as ***non-SP edges*** and all other types of edges as ordinary ***SP*** edges.

By convention, we shall assume that whenever we have a spawn or create node, the left branch leads to the spawned subroutine / created future task, whereas the right branch leads to the continuation. Thus, a serial one-core execution of the computation effectively perform a left-to-right depth-first traversal of the computation dag. Moreover, the serial one-core execution is one of the many legal schedules that can arise from parallel executions.

Structured Future

In the literature, researchers have examined how by imposing certain restrictions on the use of futures can allow for better bounds on cache misses [19], scheduling overhead [35], and race detection overhead [40]. Specifically, researchers have examined the use of ***structured futures*** which imposes the following restriction: a) ***single-touch:*** `get` is invoked on a future handle h at most once; b) ***no race on a future handle:*** there is a sequential dependence from the program point where a future handle h is created (via `create`) to the program point where a `get` is invoked on h without going through the created future task associated with h. Put it differently, given a pair of create node to its corresponding get node (by invoking `get` on the corresponding handle), there is a directed path

[5]One can easily extend the algorithm to also handle pipeline parallelism, as a race detector can handles both fork-join and pipeline parallelism similarly, as storing two total orderings of nodes suffice to perform reachability queries, but we omit pipeline parallelism here for simplicity.

[6]For simplicity, this construction described series-parallel dags with at most two incoming and two outgoing edges for each node; but can be easily generalized.

from the create node to the get node where the path starts from the continuation edge. Note that, given this restriction, it follows that a program that utilizes only `spawn`, `sync`, and structured futures can execute sequentially on one core (which follows the left-to-right depth-first traversal) without ever block on `sync` or `get`. Moreover, such program generates a class of dags that we refer to as ***SF-dags***. As we will see in Section 3, these dags have particular structural properties that can be exploited to perform race detection more efficiently.

WSP-Order

We briefly summarize how WSP-Order, the state-of-the-art race detection algorithm for SP dags [39] works here, as we utilize a similar strategy to maintain series-parallel relationship among nodes in SF-dags.

WSP-Order uses a pair of ***order-maintenance (OM)*** data structures to perform reachability analysis of the SP-dag. By storing executed nodes in two different total orderings and comparing the relative ordering of nodes in the two orders, one can tell if the two nodes are logically in parallel. Since OM data structures need to be rebalanced from time to time in order to provide constant query time, WSP-Order incorporates additional runtime support to enable parallel rebalancing with coordinated concurrent accesses to the OM data structures such that queries can be achieved with amortized constant overhead. Given an SP-dag with work T_1 and span T_∞, WSP-Order runs in time $O(T_1/P + T_\infty)$ on P cores. The runtime support is crucial in achieving this optimal running time.

3 The SF-Order and its Correctness

This section presents the full detail of SF-Order and its correctness proof. Recall that a race detector consists of two components — reachability analysis and access history. The access history remembers the necessary previous accessors per memory location. Upon a memory access v, the detector checks with the access history to find any conflicting previous access, say u. Then, the detector performs reachability query to see if there is a path from node u to v.

In this section, we will start by building some intuition about what data structures can help us answer the reachability queries, describe the full query algorithm, prove its correctness, and finally discuss why for race detecting structured futures, storing only $2k$ number of previous readers per memory location in access history suffice to perform race detection correctly, where k is the total number of futures used in the computation.

Notation

We begin with some notations. An SF-dag is generated by a set of futures which can call `create` to create a new future and call `get` on future handles in a structured manner. Each future in itself is a series-parallel (SP) dag. Therefore, an SF-dag $\mathcal{D}$ can be decomposed as a set of SP-dags connected via non-SP edges. We call each individual SP-dag $F \in \mathcal{D}$ a ***future dag*** or a ***future***. We assume that each future has a unique identifier. In addition, we say that a node $u \in F$ if u is in the SP-dag that F denotes — in this case, the instructions associated with u are part of the execution of that future. Since each future is an SP-dag, it has a unique ***first*** node which precedes all other nodes and a unique ***last*** node that all other

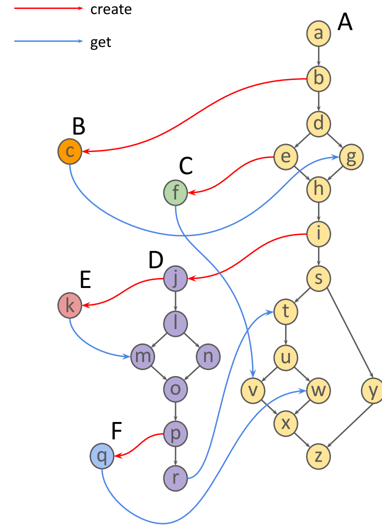

Figure 1: An example of an SF-dag.

nodes in the SP-dag precede.[7] We say that the first node of a future F is $\texttt{first}(F)$ and the last node of F is $\texttt{last}(F)$. An example SF-dag is shown in Figure 1.

We say that a future F is a ***parent*** for future G (denoted by $F = \texttt{fparent}(G)$ if some node $u \in F$ created the future G. In our example, A is the parent of B, C and D while D is E's and F's parent. Similarly, we say $F \in \texttt{f-ancs}(G)$ if F is either G's parent or parent of its parent and so on recursively, and $G \in \texttt{f-descs}(F)$ if $F \in \texttt{f-ancs}(G)$.

We can classify edges in 3 categories: ***create edges*** go from the strand $u \in \texttt{fparent}(G)$ that called $g = \texttt{create}(G)$ to $\texttt{first}(G)$ (all red edges in Figure 1); ***get edges*** go from $\texttt{last}(G)$ to the the (unique) strand that calls $\texttt{get}(g)$ where g is the future handle associated with G (all blue edges in Figure 1); and ***SP*** edges are all other edges. The create and get edges are also collectively called ***non-SP*** or NSP edges. Broadly, SP edges are edges between two nodes of the same future while NSP edges are between two nodes of different futures.

Given two nodes u and v, we use $u \to v$ to denote that there is an edge from u to v and we will sometimes use subscripts such as $u \to_{get} v$ to denote that the corresponding edge is a get edge, for example. We use $u \rightsquigarrow v$ to denote the presence of a directed path from u to v. We use $u \rightsquigarrow_{SP} v$ if *at least one* path from u to v contains only SP edges and $u \rightsquigarrow_{NSP}$ if *all* paths from u to v contain at least one NSP edge. Note that if there are multiple paths from u to v and any one of them contains only SP edges, we say that $u \rightsquigarrow_{SP} v$. We say $u \prec v$ iff $u \rightsquigarrow v$, and $u \preceq v$ iff either $u \prec v$ or $u = v$.

3.1 Intuition behind the Query Algorithm

We will start by building some intuition on how the algorithm works. Recall that race detection depends on a reachability query — given two nodes u and v, we want to answer the question: is there a path from u to v. We will consider three cases:

[7] For future tasks, we call these nodes first and last as opposed to source and sink (for ordinary SP-dags) because a future task invoking `create` can have an escaping edge leaving the dag.

(1) If $u, v \in F$ — meaning both u and v belong to the same future dag: In this case, (as we will argue in Lemma 3.3) it is sufficient to check if $u \rightsquigarrow_{SP} v$ since $u \prec v$ iff $u \rightsquigarrow_{SP} v$. Note that there may also be non-SP paths between them, but at least one path will contain only SP edges. For instance, in our example dag, even though there are non-SP paths from e to u, there is also an SP path.
(2) If $u \in F$ and $v \in G$ where $F \neq G$, but $F \in$ `f-ancs`(G): In this case, it is sufficient to check if there is a path from u to v that contains only create edges and SP edges. That is, (as we will argue in Lemma 3.5) if there is no such path, then $u \not\prec v$. Again, there may be paths from u to v that go through get edges, but at least one path will not contain any get edges. In our example, consider nodes i and q for instance.
(3) If $u \in F$ and $v \in G$ where $F \notin$ `f-ancs`(G): In this case, (as we will argue in Lemma 3.4) it is sufficient to check if there is a path from `last`(F) to v. There is a path from `last`(F) to another node u iff, for all nodes $u \in F$, we have $u \prec v$.

In our query algorithm, we separately consider these three cases. For Case 1, we can rely on asymptotically optimal race detection algorithms for series-parallel dags (such as WSP-Order [39]) since we are concerned with series-parallel dependences. For Case 3, we will rely on the idea from Xu et al.'s [43] work on race detecting general futures. In their work, for every node v, they maintain a hash table that contains all the nodes u such that $u \rightsquigarrow_{NSP} v$. However, as mentioned above, structured futures have the special property that if $u \in F$, $v \in G$, and $F \notin$ `f-ancs`(G), then $u \rightsquigarrow_{NSP} v$ iff `last`$(F) \rightsquigarrow v$. Therefore, unlike for general futures, we need not keep all such nodes u which have non-SP paths to v. Instead, for every node v, we maintain a hash table, denoted by $gp(v)$, of future IDs for all futures F where `last`$(F) \rightsquigarrow_{NSP} v$. In our example, for instance, $gp(o)$ contains B and E.

It turns out that Case 2 is the trickiest. It only applies when $F \in$ `f-ancs`(G). Therefore, for all futures G, we maintain a hash table, denoted by $cp(G)$, which contains all its ancestor futures. When checking whether $u \prec v$, we first find F and G where $u \in F$ and $v \in G$. If $F \notin cp(G)$, then this case doesn't apply. However, if $F \in cp(G)$, we have a further check. In particular, not all nodes in an ancestor future precede v — for instance, in our example, even though A is C's ancestor, $i \not\prec f$.

For this case, we will use an additional "conceptual" structure called ***pseudo-SP-dag***. A pseudo-SP-dag for an SF-dag $\mathcal{D}$, denoted by $PSP(\mathcal{D})$, is a series-parallel *approximation* of $\mathcal{D}$ which is the dag generated if we convert all `create` calls with `spawn` calls and remove all `get` calls but include an implicit `sync` at the end of a future task. Clearly, it is a series-parallel dag, since the only parallel constructs are `spawns` and `syncs`. The pseudo-SP-dag for our example is shown in Figure 2. We will say $u \twoheadrightarrow v$ iff there is a path from u to v in $PSP(\mathcal{D})$. Since $PSP(\mathcal{D})$ is a series-parallel dag, we can check if there is a path from u to v in parallel using WSP-Order [39].

This $PSP(\mathcal{D})$ itself is not sufficient to check races. Pseudo-SP-dags are inaccurate for detecting races in two ways. First, they miss some paths. First, it can be the case that $u \rightsquigarrow_{NSP} v$ while $u \not\twoheadrightarrow v$; for example, even though $j \prec u$ in $\mathcal{D}$ in Figure 1, it is not the case in the pseudo-SP-dag in Figure 2. Second, and more insidiously,

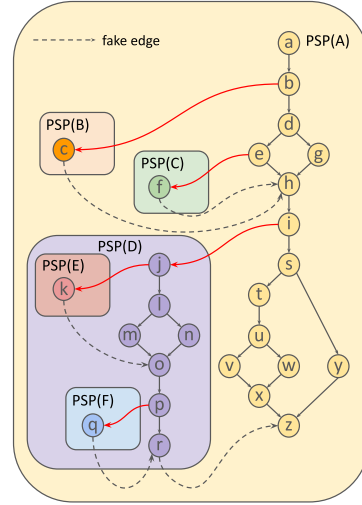

Figure 2: The corresponding pseudo-SP-dag for the SF-dag shown in Figure 1.

$PSP(\mathcal{D})$ can have ***phantom*** paths — paths that do not exist in $\mathcal{D}$. It can be the case that $u \twoheadrightarrow v$ even though $u \not\rightsquigarrow v$ in $\mathcal{D}$. For instance, in our example, $PSP(\mathcal{D})$ has a path from f to t even though such a path does not exist in $\mathcal{D}$. However, we do not use pseudo-SP-dags to check all races. Recall that we only need to use $PSP(\mathcal{D})$ to check reachability from $u \in F$ to $v \in G$ if $F \in$ `f-ancs`(G). As we will argue in Lemma 3.9, if $F \in$ `f-ancs`(G), then for all $u \in F$ and $v \in G$, we have $u \twoheadrightarrow v$ iff $u \rightsquigarrow v$.

3.2 Reachability Queries in SF-Order

We can now describe the complete query algorithm. As mentioned above, in order to perform reachability queries between nodes $u \in F$ and $v \in G$, SF-Order keeps three structures.

- Order-maintenance (OM) data structures for keeping track of series-parallel relations of $PSP(\mathcal{D})$ (similar to that in WSP-Order [39] as discussed in Section 2). This is used when $F \in$ `f-ancs`(G), both when $F = G$ (Case 1) and when F is a strict ancestor (Case 2). Intuitively, this data structure is used to check the existence of paths that either (1) contain only SP edges when $F = G$; or (2) contain only create edges and SP edges when $F \in$ `f-ancs`(G).
- For each future G, $cp(G)$ is a hash table that contains the IDs of all future ancestors of G to check if $F \in$ `f-ancs`(G) so we can use $PSP(\mathcal{D})$ for Case 2.
- For each node $v \in G$, $gp(v)$ is a hash table that contains the IDs of all futures F such that `last`$(F) \rightsquigarrow_{NSP} v$ to answer queries for Case 3.

Using these data structures, the code for a query is shown in Algorithm 1. Line 2 indicates the complete query when u and v are in the same future dag. In Lemma 3.3 and Lemma 3.7 we will argue that, in this case, it is sufficient to check if there is a path from u to v in the pseudo-SP-dag. Next, Lines 4 shows the case when $F \in$ `f-ancs`(G). In this case, we check if $u \twoheadrightarrow v$ and return true if so, which is proven to be correct in Lemma 3.9. At this point, we have already answered the query correctly if $F \in$ `f-ancs`(G). Finally, in

Lines 6, we check if $F \in gp(v)$. If so, we know that $\texttt{last}(F) \rightsquigarrow v$, which we prove in Lemma 3.4.

Algorithm 1: Reachability Query

```
1  Function Precedes(u, v)
2      if u, v ∈ F AND u ↠ v then
3          return TRUE
4      else if u ∈ F; v ∈ G AND F ∈ cp(G) AND u ↠ v then
5          return TRUE
6      else if u ∈ F; v ∈ G AND F ∈ gp(v) then
7          return TRUE
8      else
9          return FALSE
10 end
```

3.3 Correctness Proof of the Query Algorithm

We will now prove the correctness of this algorithm based on the intuition described above. First we start by some important structural properties of SF-dags.

Structural Properties of SF-dags

We start by stating some straightforward properties of SF-dags (really for any dags with futures) — these just say that paths from one future to another must go through create and/or get edges and that the only incoming create edge is into $\texttt{first}(F)$ and the only outgoing get edge is from $\texttt{last}(F)$.

Property 1. *If $u \in F$ and $v \in G$ where F and G are distinct, then any path from u to v must contain at least one non-SP edge.*

Property 2. *Among all the nodes in F, only $\texttt{first}(F)$ has an incoming create edge (other nodes may have outgoing create edges) and only $\texttt{last}(F)$ has an outgoing get edge (other nodes may have incoming get edges).*

We now restate a couple of results shown by Utterback et al. [40]. In particular, these structural properties (unlike the ones stated above) are not true for general futures. They are properties that depend on the fact that SF-dags are generated by a structured use of futures. The following lemma is implicit in Utterback et al.'s paper [40], though not stated explicitly. In particular, in their paper, they perform race detection sequentially using a left-to-right depth-first execution and this execution satisfies the following property. Their algorithm and analysis crucially depends on this property of SF-dags.

Lemma 3.1. *There is some valid execution of an SF-dag such that all future descendents of F (that is, all $G \in \texttt{f-descs}(F)$) complete execution before F completes execution.*

While the model and terminology in that paper is slightly different, the following result is a straightforward restatement of Lemma 1 in Utterback et al.'s paper [40].

Lemma 3.2. *If $u \rightsquigarrow_{NSP} v$, then there exists at least one path from u to v that contains two sections: The first path (possibly empty) contains only get edges and SP edges and the second part (possibly empty) contains only create edges and SP edges. In other words there is never a create edge followed by a get edge on this path.*

We will consider any such path from u to v to be a ***canonical path***. In Utterback et al. [40]'s model, there is a unique canonical path because they assume that the computation utilizes only structured futures but no spawns and syncs. In our model, there can be many canonical paths due to the use of spawns and syncs. For instance, in Figure 1, if we look at nodes c and q, there are multiple paths. There is a non-canonical path $c \rightarrow_{get} g \rightarrow h \rightarrow i \rightarrow_{create} j \rightarrow_{create} k \rightarrow_{get} m \rightarrow o \rightarrow p \rightarrow_{create} q$. However, we can choose not to go through future E and get the path $c \rightarrow_{get} g \rightarrow h \rightarrow i \rightarrow_{create} j \rightarrow l \rightarrow m \rightarrow o \rightarrow p \rightarrow_{create} q$ There is also another canonical path that goes through n instead of m.[8]

Case 1: $u, v \in F$. We first consider the (easy) case when $u, v \in F$ for some future F. The following lemma says that it is sufficient to check for SP paths. Note that this is distinct from general futures where $u \rightsquigarrow_{NSP} v$ even when u and v are in the same future.

Lemma 3.3. *If $u \prec v$ where $u, v \in F$, then $u \rightsquigarrow_{SP} v$.*

Proof. This property is a direct consequence of Lemma 3.2. Consider any path from u to v. If this path only contains SP edges, then we are done. Say this path does contain non-SP edges. Wlog, this path $\pi = u \rightsquigarrow_{SP} w \rightsquigarrow_{NSP} x \rightsquigarrow_{SP} v$ where the outgoing edge from $w \in F$ is the first non-SP edge on the path and the incoming edge to $x \in F$ is the last non-SP edge. Since $\texttt{last}(F)$ is the last node of F to execute in any execution, w is not $\texttt{last}(F)$. Therefore, the outgoing edge from w must be a create edge, since only the $\texttt{last}(F)$ has an outgoing get edge (Property 2). Similarly, x is not $\texttt{first}(F)$ — therefore, the incoming edge to F must be a get edge. Therefore, all paths from u to v that contain non-SP edges have a get edge after a create edge. Therefore, by the converse of Lemma 3.2, $u \rightsquigarrow_{SP} v$. □

Case 3: $u \in F$; $v \in G$; $F \notin \texttt{f-ancs}(G)$. We now consider the case where $F \notin \texttt{f-ancs}(G)$ and argue that $gp(v)$ — the hash table that contains future F iff $\texttt{last}(F) \prec v$ is sufficient to check reachability in this case.

Lemma 3.4. *If $u \rightsquigarrow_{NSP} v$ where $u \in F$ and $v \in G$ and $F \notin \texttt{f-ancs}(G)$, then $\texttt{last}(F) \rightsquigarrow v$.*

Proof. First, we argue that if $F \notin \texttt{f-ancs}(G)$, then all paths from u to v contain at least one get edge. This is easy to see from the structure of SF-dags. SP edges only connect nodes within the same future and create edges only connect futures to descendent futures. Therefore, any path from u to v where v is not in a descendent of the future containing u must go through at least one get edge.

Now consider any canonical path p from u to v — it must contain at least one get edge. We decompose p into $u \rightsquigarrow_{SP} w \rightarrow_{NSP} x \rightsquigarrow v$ where the edge from w to x is the first non-SP edge on this path. By Lemma 3.2, this first non-SP edge must be a get edge. From Property 1, we know that $w \in F$ since the path from u to w only contains SP edges. Therefore, due to Property 2, $w = \texttt{last}(F)$, since $\texttt{last}(F)$ is the only node in F with an outgoing get edge. Therefore, $\texttt{last}(F) \rightsquigarrow v$. □

[8] It turns out that the sequence of get and create edges on all canonical paths is the same. However, this property is not crucial for our proof.

Therefore, when checking reachability from $u \in F$ to $v \in G$ when $F \notin$ f-ancs(G), it is sufficient to check if last(F) has a path to v, which is exactly the information stored in $gp(v)$.

Case 2: $u \in F$; $v \in G$; $F \in$ f-ancs(G). We now consider two nodes $u \in F, v \in G$ and argue the assertion stated in Case 2 — namely, if $u \rightsquigarrow v$ and $F \in$ f-ancs(G), there is a path from u to v which contains only create and SP edges.

Lemma 3.5. *If $u \rightsquigarrow_{NSP} v$ where $u \in F$ and $v \in G$ and $F \in$ f-ancs(G), then there is at least one path from u to v containing only create and SP edges.*

Proof. Assume, for contradiction, that when $F \in G$, there is no path from u to v containing only create and SP edges — that is, there is at least one get edge on every path. By Lemma 3.2, there must be at least one path p that has all the get edges before all the create edges; and in particular, the first non-SP edge on this path must be get edge. Decompose this path into $u \rightsquigarrow_{SP} w \rightarrow_{get} x \rightsquigarrow v$. From Property 1, x is the first node on this path that is not in F and from Property 2, $w =$ last(F) since that is the only node in F that has an outgoing get edge.

From Lemma 3.1, there is some execution S where G finishes executing before F finishes execution. Therefore, there cannot be a path from $w =$ last(F) to $v \in G$. Hence a contradiction. □

Therefore, when $F \in$ f-ancs(G), we must somehow check for the existence of a path that contains only create and SP edges. This is where the pseudo-SP-dag $PSP(\mathcal{D})$ comes in. Recall that we simply convert all creates into spawns, remove all get statements, and include implicit syncs to generate $PSP(\mathcal{D})$. We say $u \twoheadrightarrow v$ if there is a path from u to v in the pseudo-SP-dag. We now argue that $PSP(\mathcal{D})$ precisely answers queries between $u \in F$ and $v \in G$ if $F \in$ f-ancs(G).

For convenience, we will define $PSP(F)$ for all futures F in a similar manner — the entire SP-subdag generated by F which has first(F) as the first node and last(F) as the last node is called $PSP(F)$. The following lemma is true due to the construction since all create edges are converted to spawn edges.

Lemma 3.6. *For any node $v \in G$, $v \in PSP(F)$ iff $G \in$ f-descs(F) (including F)*

In our example, all nodes are part of $PSP(A)$ while nodes from E and F are part of $PSP(D)$ since they are both D's descendents.

Let us consider some relationships between paths in SF-dags and the corresponding pseudo-SP-dags. The following lemma considers $u, v \in F$ and says that $PSP(F)$ precisely denotes the relationship between such nodes. This justifies our decision in Line 2 to simply check the pseudo-SP-dag when checking if $u \prec v$ when u and v are in the same future F.

Lemma 3.7. *If $u, v \in F$, then $u \twoheadrightarrow v$ iff $u \prec v$*

Proof. From Lemma 3.3, we know that if $u \prec v$ then there is an SP path between u and v. We do not remove any SP paths in the pseudo-SP-dags. Therefore, this path cannot be removed. Conversely, if there is no path between u and v, then they are in two separate SP-subdags of F. More precisely, there is node s such that u is in the left subdag of s and v is in the right subdag (or vice-versa). Therefore, in the pseudo-SP-dag, this relationship between u and v will still hold. Since pseudo-SP-dag is an SP-dag, there can be no paths between a node in the left subdag and a node in the right subdag of s just due to the properties of SP-dags. □

The next lemma states that pseudo-SP-dags are also good at finding paths that contain only create and SP edges, since the only edges removed are get edges.

Lemma 3.8. *If $u \in F$, $v \in G$ where $F \in$ f-ancs(G) and $u \prec v$, then $u \twoheadrightarrow v$.*

Proof. From Lemma 3.5, at least one path from u to v has no get edges and consists of only SP edges and create edges. Since the pseudo-SP-dag construction does not remove any SP or create edges, this path would still exist in $PSP(\mathcal{D})$. □

However, this in itself is not sufficient to precisely detect races since it is not obviously an if and only if statement. In particular, we might worry that $u \twoheadrightarrow v$ even if $u \not\rightsquigarrow v$. For instance, in our example, $PSP(\mathcal{D})$ has a path from f to t even though such a path does not exist in the original dag $\mathcal{D}$. These phantom paths are due to ***fake edges***, denoted by $\rightarrow_{fake}$. In particular, pseudo-SP-dags have additional sync edges that are not in the original SP-dag — these are the get edges from the last node of a child future G to a sync node in the parent future F. In our example, the offending fake edge is from f to h. We will say a path from u to v is fake (denoted by $u \twoheadrightarrow_{fake} v$) if $u \twoheadrightarrow v$, but $u \not\rightsquigarrow v$. Clearly, a fake path must have one or more fake edges.

The following lemma says that, even though there can be fake paths in pseudo-SP-dags, they do not occur between $u \in F, v \in G$ if $F \in$ f-ancs(G).

Lemma 3.9. *If $u \in F; v \in G$ such that $F \in$ f-ancs(G), then $u \twoheadrightarrow v$ implies $u \rightsquigarrow v$.*

Proof. Assume for contradiction that $u \twoheadrightarrow_{fake} v$ — that is, all paths from u to v contain at least one fake edge. Consider the path p with the smallest number of these fake edges.

Due to the way the pseudo-SP-dag is constructed, a fake edge always goes from last(H) for some future H to some sync node in fparent(H).

Say $X = \{F_1 = F, F_2, F_3, ..., F_k = G\}$ be the set of all the futures which are ancestors of G but not ancestors of F in order of depth in the create-tree. That is, if we look at the create-tree, these are the futures on the path from F to G.

Case 1: Some fake edge on p goes from last(H) to some node $y \in$ fparent(H) where $H \notin X$. In this case, the path p can be decomposed to $u \twoheadrightarrow x \rightarrow_{create}$ first$(H) \twoheadrightarrow$ last$(H) \rightarrow_{fake} y \twoheadrightarrow v$ where $x, y \in$ fparent(H) since both $PSP(H)$ and $PSP($fparent$(H))$ are series-parallel dags. In particular, all paths to last(H) must go through first(H) (unless they originate in this subdag). In addition, there must be a path directly from x to y that uses only edges within fparent(H) since there is always a path from the create (spawn) node to the corresponding sync node. Therefore, we can replace the subpath with fake edge with a subpath without fake edge, contradicting the minimality assumption.

Case 2: All fake edges on the path p go from last(F_{i+1}) to $y \in F_i$. Therefore, there is a path $u \twoheadrightarrow$ last$(F_{i+1}) \rightarrow_{fake} y \twoheadrightarrow v$. However, by Lemma 3.6, $G \in PSP(F_{i+1})$. Therefore, there is a path from v to

$\texttt{last}(F_{i+1})$ since all nodes within a series-parallel (sub)dag have a path to the last of that series-parallel (sub)dag. However, we cannot have $\texttt{last}(F_{i+1}) \twoheadrightarrow v \twoheadrightarrow \texttt{last}(F_{i+1})$ since $PSP(F_{i+1})$ is a dag. ◻

3.4 Maintaining the Reachability Data Structures On-the-fly

As mentioned in Section 3.2, SF-Order maintains three separate data structures: (1) A reachability data structure for the pseudo-SP-dag. (2) The $gp(v)$ hash table — for every node v, this table has the IDs off all futures F such that $\texttt{last}(F) \prec v$. (3) The $cp(G)$ hash table — for every future G, it stores the IDs of all future ancestors of G.

We now briefly explain how these data structures are maintained during a parallel execution. To check reachability within the pseudo-SP-dag, we use WSP-Order described by Utterback et al. [39]. The $cp(G)$ data structures is also easy to maintain. When a future G is created by future H, it simply copies over its parent's hash table ($cp(H)$) and adds its own ID to it. Maintaining $gp(v)$ is slightly more complicated, but not by much — the argument is identical to the one given by Xu et al. [43]. Conceptually, a node simply gets the union of its parent's tables — $gp(v) = \cup_{u \to v} gp(u)$. Since we cannot afford to copy hash tables at every new node — we use pointers most of the time. If a node has a single parent, it need simply keep a pointer to its parents hash table and refer to it directly. We need only create new hash tables when a node has multiple parents and their tables must be merged — that is, at sync nodes and at get nodes. Naively merging at every sync and get is also too expensive — while there are only k get nodes in the computation (one for each future), there could be many more sync nodes. We can be cleverer about the implementation however, and only perform a merge if among the hash tables associated with the two parents of a node, each contain some item that the other does not contain. Xu et al. [43] argue that this can occur at most k times during the computation and that argument holds here as well.

3.5 The Access History Component

In a race detection algorithm, the access history stores the readers and writers that previously accessed a given memory location. For programs with only fork-join parallelism (i.e., SP-dags), given a memory location l, Mellor-Crummey [26] has shown that it suffices to store one previous writer — called ***last writer***, that is simply the last writer that wrote to l, and two readers — the ***rightmost reader*** $\texttt{rreader}(l)$ and the ***leftmost reader*** $\texttt{lreader}(l)$. For programs with general futures, however, the race detector must store an arbitrarily large number of previous readers for each memory location [1].

By exploiting the restrictions imposed by structured futures, we show that one can store only $2k$ readers per memory location, where k is the total number of futures used in the computation, without breaking the correctness guarantees. In particular, given a memory location l and a future dag F in an SF-dag $\mathcal{D}$, SF-Order stores only the rightmost reader $\texttt{rreader}(l, F)$ and leftmost reader $\texttt{lreader}(l, F)$ of l with respect to F (that is, the leftmost and rightmost readers of l in F compared to all other readers of l in F). Recall from Lemma 3.3, if two nodes u and v are in the same future dag and $u \prec v$, then there must exist an SP path between them. Thus the following lemma straightforwardly follows from prior work by Mellor-Crummey [26].

Lemma 3.10. *At any point during the execution of an SF-dag, let $R_{(l,F)}$ be the set of nodes in future dag F that have read memory location l and w be any other node in F. We have $r \prec w$ for all $r \in R_{(l,F)}$ iff $\texttt{rreader}(l, F) \prec w$ and $\texttt{lreader}(l, F) \prec w$.*

Lemma 3.10 says that given a memory location l and a future F, storing its rightmost and leftmost readers suffice to detect intra-future races. Now we prove these readers also suffice to detect inter-future races.

Lemma 3.11. *At any point during the execution of an SF-dag, let $R_{(l,F)}$ be the set of nodes in a future dag F that have read memory location l and w be any other node in some future G distinct from F. We have $r \prec w$ for all $r \in R_{(l,F)}$ iff $\texttt{rreader}(l, F) \prec w$ and $\texttt{lreader}(l, F) \prec w$.*

Proof. If all $r \in R_{(l,F)}$ precede w then $\texttt{rreader}(l, F)$ and $\texttt{lreader}(l, F)$ must also precede w since they are in the set $R_{(l,F)}$.

Now we show the other direction — assuming that $\texttt{rreader}(l, F) \prec w$ and $\texttt{lreader}(l, F) \prec w$, we need to show that all $r \in R_{(l,F)}$ precede w. Since $w \notin F$, we have $\texttt{rreader}(l, F) \rightsquigarrow_{NSP} w$ and $\texttt{lreader}(l, F) \rightsquigarrow_{NSP} w$ (Property 1). Let's first consider the case where $F \notin \texttt{f-ancs}(G)$. Then by Lemma 3.4, $\texttt{last}(F) \rightsquigarrow w$, and thus all nodes in F precede w.

Next we consider the case that $F \in \texttt{f-ancs}(G)$. Since both $\texttt{rreader}(l, F)$ and $\texttt{lreader}(l, F)$ are in F, $w \in G$, and $F \in \texttt{f-ancs}(G)$, by Lemma 3.5, there is at least one path from $\texttt{rreader}(l, F)$ to w containing only create and SP edges. We can decompose this path into $\texttt{rreader}(l, F) \rightsquigarrow_{sp} x \to_{create} y \rightsquigarrow z \to_{create} \texttt{first}(G) \rightsquigarrow_{sp} w$ (where $y \rightsquigarrow z$ can be empty if F is the immediate future parent of G). Similarly with the same argument we can decompose the path from $\texttt{lreader}(l, F)$ to w into $\texttt{lreader}(l, F) \rightsquigarrow_{sp} x' \to_{create} y' \rightsquigarrow z' \to_{create} \texttt{first}(G) \rightsquigarrow_{sp} w$ (where $y' \rightsquigarrow z'$ can be empty if F is the immediate future parent of G). Since each future has exactly one parent, we have $z = z'$ and $x = x'$ inductively. Therefore, we have $\texttt{rreader}(l, F) \rightsquigarrow_{sp} x$ and $\texttt{lreader}(l, F) \rightsquigarrow_{sp} x$. Then based on Lemma 3.10, we know for any other reader r in $R_{(l,F)}$, r must precede x as well, which leads to $r \prec w$. ◻

3.6 Performance Analysis of SF-Order

Now we can analyze the performance bound for SF-Order. First, we can state the following bound for the reachability component based on the construction discussed in Section 3.4:

Lemma 3.12. *Given a computation with work T_1, span T_∞, and k futures, constructing the reachability data structure has total work of $O(T_1 + k^2)$ and total span of $O(T_\infty + \min\{T_\infty, k\} \lg k)$. Therefore, the running time on P processors is $((T_1 + k^2)/P + T_\infty + \min\{T_\infty, k\} \lg k)$.*

Proof. Maintaining WSP-Order to answer reachability queries between $PSP(\mathcal{D})$ has no asymptotic overhead [39]. $cp(G)$ for each future is constructed when the future is created and takes at most k extra work, for a total of k^2 overhead for k futures. As for $gp(v)$, we argued that new hash tables are created at most $O(k)$ times — once for each of the k get node and at k sync nodes at the most.

Each of these merges takes $O(k)$ time since no hash table can be larger than k. Therefore, the total work is $O(T_1 + k^2)$.

Every copy and merge can be done in parallel for the span of $O(\lg k)$. Since there are at most k such merges, this overhead on the span can not be larger than $O(k \lg k)$. In addition, at most T_∞ of these merges fall along the critical path — hence the result. □

We can also show easily that queries are cheap. Utterback et al. [39] show that WSP-Order answers queries in $O(1)$ time amortized. In addition to that, we only check $gp(v)$ and $cp(G)$ once for each query, which each take $O(1)$ time in expectation.

Lemma 3.13. *Checking if $u \prec v$ using the query algorithm takes $O(1)$ time amortized and in expectation.*

Now we can state the final performance bound for SF-Order:

Theorem 3.14. *Given a computation with work T_1 and span T_∞, SF-Order executes in time $O((T_1 + k^2)/P + T_\infty \lg k)$ on P processing cores, where k is the total number of futures used in the computation.*

Proof. On a read, the the race detector has to check races against at most one previous writer and each query takes $O(1)$ time. On a write, the race detector may check races against at most $2k$ previous readers. Therefore, each write may cause up to $O(k)$ work and $O(\lg k)$ span (since all these checks can be done in parallel). However, these reads can then be removed from the access history; therefore, this $O(k)$ work can be amortized against the cost of performing them in the first place. By Lemma 3.12, the reachability structure construction costs $O(k^2)$ asymptotic overhead giving us the work term. For the span, the $O(\lg k)$ overhead is multiplicative on the span; therefore, the additive overhead of construction is absorbed by it, leaving us with the total span of $O(T_\infty \lg k)$. The P processor bound follows from standard scheduling theorems. □

4 Implementation and Empirical Evaluation of SF-Order

We have implemented SF-Order and empirically evaluated it by comparing it against MultiBags [40],[9] the state-of-the-art sequential race detector designed for structured futures, and F-Order [43],[10] the state-of-the-art parallel race detector designed for general futures. Experimental evaluation indicates that, although our algorithm incurs a higher overhead for one-core executions compared to MultiBags, its absolute running time wins out when running on two cores or more as MultiBags can only run the program sequentially. On the other hand, when compared F-Order, our algorithm incurs lower overheads in general, due to the lower reachability construction and query overheads.

Implementation Overview

Here we briefly describe the implementation of SF-Order. As discussed in Section 3, the reachability component of SF-Order requires three different types of data structured. The first one is the SP-Order data structure from the WSP-Order algorithm [39] to maintain the series-parallel ordering of pseudo-SP-dags. The WSP-Order algorithm requires a specialized runtime system support in order to obtain the amortized constant time query overhead on SP-Order. Such runtime system support is similarly required by F-Order [43], and thus in their framework, they provided an extended Cilk-F runtime system [35], a work-stealing runtime system that supports the use of futures and the specialized runtime system support for WSP-Order. We have taken this extended Cilk-F runtime and incorporated into our software that implements SF-Order.

For gp, recall that it is simply a hash table per node v in the SF-dag that keeps track of the IDs of all futures F such that the last node of F is an ancestor of v. One bit suffices to store such information per unique future in the execution. Thus, instead of utilizing an actual hash table hashing the unique IDs of such futures F, we utilized an array of 64-bit integers to indicate membership of $gp(v)$ — a bit in position i indicates whether the last node of a future F with ID i is an ancestor of v.

Similarly for cp, it is again a hash table containing the IDs of all future ancestors F of a given future G. Again, one bit suffices to store such information per unique future F. Thus, we similarly utilized an array of 64-bit integers to indicate membership of $cp(G)$ instead of an actual hash table.

Finally, for the access history component, we utilized the same access history construction as in F-Order — a two-level hash table that acts like a direct-mapped cache, hashing the address of a memory location to its metadata. Even though SF-Order can bound the number of readers per memory location in the access history, doing so required that we utilize yet another hash table in the metadata for a given memory location, which hashes from a future ID to its leftmost and rightmost reader. Since the overall space, hashing, and additional query overhead (to check if some reader is the leftmost or right most compared to existing readers) likely outweigh the saving in the number of readers we can omit, we simply store all the readers in the hash table between writes like what was done in F-Order.

Experimental Setup

All experiments were conducted on a machine with two 20-core Intel Xeon Gold 6148 cores, clocked at 2.40 GHz. Hyperthreading and dynamic frequency scaling are disabled. Each core has a separate private L1 data cache and L1 instruction cache, with 32KB capacity each. Each core also has a 1MB private L2 cache. Each socket has a 27.5 MB L3 cache shared among 20 cores. The machine has 768 GB of main memory. We have used only one socket for the experiments to avoid variance due to NUMA effect. All software is compiled with LLVM/Clang 3.4.1 with `-O3` optimization level, running on Linux kernel version 4.15. Each data point is the average of five runs with standard deviation less than 5.5%.

We have used five benchmarks to evaluate performance, including divide-and-conquer matrix multiplication (`mm`), parallel mergesort (`sort`), Smith-Waterman sequence alignment (`sw`), the Heart Wall application (`hw`) from the Rodinia benchmark suite [9] that tracks the movement of a mouse heart over a sequence of ultrasound images, and the Ferret application (`ferret`) adapted from the PARSEC benchmark suite [5] that implements a content-based similarity search on images. The inputs and execution characteristics of the benchmarks are shown in Figure 3.

[9] The codebase of MultiBags can be obtained at https://github.com/wustl-pctg/futurerd.
[10] The codebase of F-Order can be obtained at https://github.com/wustl-pctg/f-order.

bench	*N*	*B*	*# reads*	*# writes*	*# queries*	*# futures*	*# nodes*
mm	2048	64	1.72×10^{10}	1.43×10^8	1.32×10^8	18724	79577
sort	10^7	8192	2.75×10^8	2.22×10^8	1.21×10^7	14463	60030
sw	2048	64	8.59×10^9	4.20×10^6	8.58×10^9	1024	2054
hw	10 (images)	-	1.73×10^{10}	1.64×10^8	1.75×10^{10}	3672	9914
ferret	simlarge	-	5.40×10^9	6.23×10^8	7.40×10^9	256	1280

Figure 3: The input size (*N*), basecase size (*B*), and execution characteristics of the benchmarks, including the total numbers of reads, writes, reachability queries performed throughout the execution, the number of futures used, and the number of nodes in the computation dag.

bench	*base* (T_1)	*base* (T_{20})	*config*	*MultiBags* (T_1)	*F-Order* (T_1)	*SF-Order* (T_1)	*F-Order* (T_{20})	*SF-Order* (T_{20})
mm	8.02	0.42 [19.10×]	*reach*	8.14 (1.01×)	11.36 (1.42×)	8.38 (1.04×)	0.64 [17.75×]	0.43 [19.49×]
			full	305.73 (37.84×)	468.59 (58.43×)	447.28 (55.77×)	23.62 [19.84×]	22.51 [19.87×]
sort	1.30	0.07 [18.57×]	*reach*	1.27 (0.99×)	3.90 (3.00×)	1.35 (1.04×)	0.33 [11.82×]	0.07 [19.29×]
			full	17.56 (13.72×)	28.44 (21.88×)	26.20 (20.15×)	2.10 [13.54×]	1.86 [14.09×]
sw	20.92	2.14 [9.78×]	*reach*	20.90 (1.00×)	24.94 (1.19×)	24.25 (1.16×)	2.15 [11.60×]	2.14 [11.33×]
			full	583.78 (27.85×)	676.39 (32.33×)	555.39 (26.55×)	73.87 [9.16×]	64.75 [8.58×]
hw	14.87	0.95 [15.65×]	*reach*	14.77 (1.00×)	15.90 (1.06×)	15.22 (1.02×)	0.99 [15.91×]	0.95 [16.02×]
			full	333.35 (22.62×)	887.59 (59.69×)	676.25 (45.58×)	62.78 [14.14×]	51.77 [13.05×]
ferret	6.84	0.73 [9.73×]	*reach*	6.70 (1.01×)	7.10 (1.04×)	6.95 (1.02×)	0.75 [9.47×]	0.71 [9.79×]
			full	278.5 (42.07×)	308.14 (45.05×)	270.70 (39.58×)	29.88 [10.31×]	25.52 [10.61×]

Figure 4: Execution times of the benchmarks shown in seconds for the baseline executions (i.e., with no race detection, shown as *base*) and when running with MultiBags, F-Order, and SF-Order for race detection with two different configurations. The first configuration shown as *reach* runs each algorithm with only the reachability construction overhead. The second configuration shown as `full` runs the full race detetion algorithm. Columns with T_1 show the execution times running on one core, and columns T_{20} show the execution times running on 20 cores. Numbers in the parentheses show the overhead compared to the baseline executions. Numbers in the brackets show the scalability relative to the T_1 time of the same configuration.

Empirical Evaluation of SF-Order

We have compared the performance of SF-Order against MultiBags and F-Order using five benchmarks described above, with the results shown in Figure 4. Specifically, we evaluated each algorithm with two different configurations — the `reach` configuration runs the applications with only the reachability maintenance without actually performing race detections on memory accesses, and the `full` configuration that runs the full race detection. The `reach` configuration incurs overhead only upon the execution of a parallel construct, and thus shows only the construction overhead for the reachability component. The `full` configuration incurs the full overhead, including the constructing the reachability, updating the access history, and performing the necessary queries into both the reachability and access history upon a memory access.

In theory , MultiBags incurs the least amount of overhead asymptotically (a multiplicative overhead in the inverse Ackermann's function, which is upper bounded by 4 for all practical purposes [10]), whereas for F-Order and SF-Order, there is an additional $O(k^2)$ overhead for the reachability construction. In practice, the reachability construction incurs rather negligible overhead for both MultiBags and SF-Order, whereas the overhead for F-Order is higher.

The reason behind SF-Order's lower overhead than F-Order in practice (despite having the same asymptotic overhead) is as follows. Like SF-Order, F-Order needs to maintain some type of hash table per node during execution (which is akin to the *gp* and *cp* data structures needed by SF-Order). However, due to the properties of SF-dags, it suffices for SF-Order to maintain a *gp* (or a *cp*) as an array of bitmaps as opposed to using an actual hash table, whereas F-Order needs to employ a full-fledged hash table per node, which incurs higher space and time overheads. We additionally measured and compared the space overhead between F-Order and SF-Order. As shown in Figure 5, SF-Order incurs significantly less space overhead, only 1.29% of the memory usage of F-Order on average, for five benchmarks.

bench	*F-Order*	*SF-Order*
mm	9.1	0.07
sort	7.64	0.05
sw	0.14	2×10^{-4}
hw	1.7	6×10^{-3}
ferret	6×10^{-3}	6.50×10^{-5}

Figure 5: Memory usage of the benchmarks when running with F-Order and SF-Order for reachability maintenance, shown in gigabytes.

The full race detection is expensive across all algorithms. This is especially evident in the T_1 running times with the `full` configuration. Both parallel algorithms F-Order and SF-Order incur higher overheads than MultiBags, with SF-Order incurs less overhead than F-Order. This is actually in large part due to the fact that, queries into the access history needs to be synchronized with locks in the parallel algorithms. Since MultiBags executes sequentially, it does

not incur such an overhead. In particular, for both F-Order and SF-Order, every time a read or a write occurs, one must acquire lock on the access history. The access history does utilize fine-grained locking (each lock represents a subset of the access history containing 16-byte memory locations), so contention is not really the issue. Rather, the high overhead stem from the sheer volume of locking operations necessary (which tracks the number of reads and writes shown in Figure 3). Compared to F-Order, SF-Order incurs lower overhead in the `full` configuration due to its lower reachability query overhead. In particular, SF-Order tends to have higher savings in overhead compared to F-Order on applications with large number of queries (e.g., `sw` and `hw`). However, the savings are dwarfed by the locking overhead. We have separately measured T_1 for F-Order and SF-Order without using locks in access history and confirmed that the locking overhead is indeed significant and dominates the additional overheads seem in `full`.

Even though F-Order and SF-Order incur higher overhead than MultiBags, they both exhibit scalability that closely tracks that of the baseline executions. Xu et al. documented that when compared with MultiBags, F-Order wins out in absolute running times as long as four or more cores are used. Since SF-Order incurs lower overheads, when compared to MultiBags, SF-Order's absolute running time wins out when two or more cores are used.

5 Related Work

History of Determinacy Race Detections

There is a long history to the problem of detecting determinacy races in task-parallel code. Prior to the work by Dimitrov et al. [12], which is the first to examine the problem for two-dimensional dags, most work has focused on race detecting series-parallel dags. The work by Nudler and Rudolph [29] was the first one to observe that storing two total ordering of nodes suffice for SP-dags — one ordering, called the ***English ordering***, traverse the dag in the left-to-right depth-first manner, and the other ordering, called the ***Hebrew ordering***, traverse the dag in the (opposite) right-to-left depth-first manner. By comparing the relative positions of two nodes in these two orderings, one can tell if the two nodes are logically in parallel with each other. Later, Mellor-Crummey [26] proposed a parallel algorithm for race detecting SP-dags and showed that storing a constant number of accessors per memory location suffices. Both of these works were not concerned about the resource bounds of the race detection algorithm, however.

Feng and Leiserson [13, 14] proposed the first provably efficient sequential algorithm for race detecting SP-dags. Their construction utilizes a union-find data structure to maintain reachability, which can be shown to incur little overhead (inverse Ackermann's function, which is upper bounded by 4 for all practical purposes [10]). However, the algorithm cannot be parallelized, as the construction depends on updating the union-find data structures with the left-to-right depth-first traversal of the computation dag.

Later, multiple provably efficient parallel algorithms for race detecting SP dags were proposed [4, 15, 39]. Generally, these works utilize the strategy of storing the executed nodes with the two total orderings proposed originally by [29] to perform reachability queries. In order to maintain and query the orderings efficiently, order-maintenance data structures are used. The distinctions among these works are how they handle the management of these OM data structures to enable efficient queries during parallel executions. In particular, as discussed in Section 2, the state-of-the-art algorithm WSP-Order [39] utilizes a specialized runtime scheduler in order to manage the rebalancing of OM data structures to achieve amortized constant-time queries.

Beyond SP-dags, the two-dimensional dags also have nice structural properties. The first algorithm, proposed by Dimitrov et al. [12] again utilizes union-find data structures to maintain reachability and therefore is fundamentally sequential. Later Xu et al. [42] proposed a parallel algorithm for race detecting two-dimensional dags using a similar strategy of storing two total orderings designed for two-dimensional dags. The way the two total orderings are maintained is similar to that in WSP-Order using OM data structures with specialized runtime system support.

More recently, the use of futures has drawn interests, and a few algorithms have been proposed to race detect programs with futures [1, 37, 40, 43]. However, since the use of futures generates arbitrary dependences, the strategy of storing two total orderings is no longer applicable. Thus, the state-of-the-art parallel algorithm by Xu et al. [43] takes a new approach of keeping track of non-SP (i.e., create and put nodes) ancestors in order to perform reachability queries. Our algorithm is similar in spirit, but our algorithm is able to exploit the additional properties enforced by structured futures to obtain lower overhead.

Bounds on Programs with Futures

Beyond bounds for race detecting programs with futures, researchers have examined other resource usage bounds on computations with futures. Spoonhower et al. [36] examined bounds on the number of cache misses and scheduling overhead incurred by a classic work stealing scheduler [2, 6] during parallel executions when the computation utilizes general futures. Later, Herlihy and Liu [19] showed that, better bounds on cache misses and scheduling overhead with work stealing can be achieved when the computation utilizes only structured futures. Later, Singer et al. [35] proposed proactive work stealing that diverges from classic work stealing [2, 6] and showed that proactive work stealing achieves better bounds on cache misses and scheduling overhead, albeit with slightly worse execution time bound.

6 Conclusion

This work examines whether one can race detect programs with structured futures more efficiently than what was known for general futures. To that end, we propose a parallel race detection algorithm that is asymptotically more efficient than the prior work designed for general futures and show that it incurs lower overhead than the prior work in practice. The experiments indicate that, however, much of the overhead stems from the need to synchronize on access history. An interesting future direction is to examine whether one can reduce the synchronization overhead by redesigning the access history.

Acknowledgments

This work was supported in part by the National Science Foundation under grant numbers CCF-1733873, CCF-1910568, CCF-1943456, CCF-1725647 and CCF-1439062.

References

[1] Kunal Agrawal, Joseph Devietti, Jeremy T. Fineman, I-Ting Angelina Lee, Robert Utterback, and Changming Xu. 2018. Race Detection and Reachability in Nearly Series-Parallel DAGs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. New Orleans, Louisiana.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* (2001), 115–144.

[3] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, Orlando, Florida, USA, 735–736.

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

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*. ACM, 72–81.

[6] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *JACM* 46, 5 (1999), 720–748.

[7] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. 51–61.

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

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). The MIT Press.

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

[12] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. 2015. Race Detection in Two Dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, Portland, Oregon, USA, 101–110. https://doi.org/10.1145/2755573.2755601

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

[14] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.

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

[16] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2010. Implicitly Threaded Parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (Nov. 2010), 537–576. https://doi.org/10.1017/S0956796810000201

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

[18] Robert H. Halstead, Jr. 1985. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS* 7, 4 (Oct. 1985), 501–538.

[19] Maurice Herlihy and Zhiyu Liu. 2014. Well-structured Futures and Cache Locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, Orlando, Florida, USA, 155–166. https://doi.org/10.1145/2555243.2555257

[20] Shams Imam and Vivek Sarkar. 2014. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag New York, Inc., New York, NY, USA, 618–643. https://doi.org/10.1007/978-3-662-44202-9_25

[21] Intel Corporation 2012. *Intel(R) Threading Building Blocks*. Intel Corporation. Available from http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm.

[22] Intel Corporation 2013. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1*. Intel Corporation. Document 324396-002US. Available from http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm.

[23] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. 2010. Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems. In *PACT*. ACM, 411–420.

[24] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *J. Supercomputing* 51, 3 (2010), 244–257.

[25] Li Lu, Weixing Ji, and Michael L. Scott. 2014. Dynamic Enforcement of Determinism in a Parallel Scripting Language. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, Edinburgh, United Kingdom, 519–529.

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

[27] Stefan K. Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 577–591.

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

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

[30] OpenMP 4.0 2013. *OpenMP Application Program Interface, Version 4.0*.

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

[32] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 531–542.

[33] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc.

[34] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2020. Priority Scheduling for Interactive Applications. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 465–477.

[35] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/3293883.3295735

[36] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. 2009. Beyond Nested Parallelism: Tight Bounds on Work-stealing Overheads for Parallel Futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, Calgary, AB, Canada, 91–100. https://doi.org/10.1145/1583991.1584019

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

[38] Olivier Tardieu, Haichuan Wang, and Haibo Lin. 2012. A Work-stealing Scheduler for X10's Task Parallelism with Suspension. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New Orleans, Louisiana, USA, 267–276.

[39] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, Asilomar State Beach, CA, USA, 83–94.

[40] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 2019. Efficient Race Detection with Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, Washington, District of Columbia, 340–354.

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

[42] Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal. 2018. Efficient Parallel Determinacy Race Detection for Two-dimensional Dags. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, Vienna, Austria, 368–380.

[43] Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. 2020. Parallel Determinacy Race Detection for Futures. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. ACM, San Diego, California, 217–231.