Parallel Determinacy Race Detection for Futures

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

Kyle Singer
Washington University in St. Louis
kdsinger@wustl.edu

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

Abstract

The use of futures can generate arbitrary dependences in the computation, making it difficult to detect races efficiently. Algorithms proposed by prior work to detect races on programs with futures all have to execute the program sequentially. We propose F-Order, the first known parallel race detection algorithm that detects races on programs that use futures. Given a computation with work T_1 and span T_∞, our algorithm detects races in time $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(k + \lg r \lg \hat{k}))$ on P processors, where k is the number of future operations, r is the maximum number of readers per memory location, and $\hat{k}$ is the maximum number of future operations done by a single future task, which is typically small. We have also implemented a prototype system based on the proposed algorithm and empirically demonstrates its practical efficiency and scalability.

*CCS Concepts• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Shared memory algorithms*; • **Computing methodologies** → Shared memory algorithms.

1 Introduction

Futures [17] provide an elegant means to express parallelism in functional programs. Using futures, the programmer can annotate the invocation of a function instance to be asynchronous — the function call may execute in parallel with the continuation of the caller. The invocation returns a ***future handle*** object that associates the function call execution, the ***future task***, with a store. Whenever the execution of a future task completes, the resulting value of the task is deposited into the future handle and can be retrieved by anyone who has access to the handle. An attempt to retrieve the result from the handle before the computation finishes causes the control to block.

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.

PPoPP'20, February 22-26 2020, San Diego, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374536

Although futures were originally proposed in the context of functional programs, its use has seen an increasing popularity in the context of imperative programs. Modern imperative task parallel platforms that support futures include variants of Habanero Java [9, 21], X10 [10, 48], Cilk [43], and Deterministic Parallel Ruby [27]. Its use has also been incorporated into C++11 [22] and Java [34]. Recently, researchers have also studied how to use futures in multithreaded computations to share concurrent data structures [24, 30].

Compared to fork-join parallelism, which is the traditional parallelism paradigm supported by many task parallel platforms, futures offer additional flexibility in how one can express parallelism. With fork-join parallelism, the asynchronous function invocations must be joined together within a well-defined lexical scope; with futures, joining with a future task occurs when one invokes ***get*** on the handle to attempt to retrieve the result of the future task. Since the future handle is an object that can be freely passed around, or even stored in a data structure and retrieved later, joining with future tasks can occur at arbitrary program points and is not limited by any lexical scope.

With imperative programming, the interactions between shared memory and parallelism have to be regarded carefully, however. If not careful, the interactions can lead to pernicious errors such as races that are challenging to detect and debug. In this paper, we study the problem of how to efficiently detect races in a task parallel program that employs futures.

How to efficiently detect races in a task parallel program is a well studied problem [1, 4, 13–15, 29, 37, 38, 47, 50, 51, 55]. In the context of task parallel programming, the focus has been on detecting ***determinacy races*** [13] (also called ***general races*** [31]), where two *logically* parallel subcomputations access the same memory location in a conflicting way (i.e., at least one is a write). The execution of a parallel computation can be modeled as a ***directed acyclic graph*** (or ***DAG***), in which a node represents a sequence of instructions without parallel control constructs and an edge represents a control dependence that arises from the execution of control constructs. Two memory accesses are said to be logically in parallel if no directed path exists between them. Since the dependences that arise from the control constructs are input-dependent and not schedule dependent, the race detection algorithms proposed by this prior work can provide strong correctness guarantees — the race detector reports a race if and only if one exists for a given program and a given input; i.e., the race detector is *sound and complete with respect to a given input*, as opposed to for a given execution

schedule. Moreover, in the absence of a determinacy race,[1] the computation behaves in a deterministic fashion.

Algorithms proposed by this prior work detect races ***on the fly*** as the program executes. They typically consist of two components: 1) an ***access history*** that keeps track of the readers and the writers that previously accessed a given memory location during execution, and 2) a ***reachability*** component that answers the query of whether two given accesses are logically in parallel or not. Upon a memory access v, the algorithm checks v against each conflicting access u stored in the access history that accessed the same memory location, and queries the reachability data structure to see if u and v are logically in parallel. If so, the algorithm has found a race.

Much of the prior work focuses on detecting races for fork-join parallelism [4, 13–15, 29, 37, 38, 50], which generates computation DAGs with nice structural properties. By exploiting the structural properties, the state-of-the-art algorithm [50] can detect races for a task parallel computation with only fork-join parallelism with no asymptotic overhead. Given a computation with ***work*** T_1, how long it takes to run the computation on one core and ***span*** T_∞, how long it takes to run the computation on infinitely-many cores, or the longest dependences in the computation, the algorithm can race detect the computation in time $O(T_1/P + T_\infty)$ when running on P processing cores.

Unlike computations that utilize only fork-join parallelism, the use of futures can generate arbitrary dependences due to the fact that the joining of future tasks can occur at arbitrary program points. Consequently, a program that uses futures no longer has the same structural properties, and algorithms developed in this prior work cannot be applied directly.

A few *sequential* algorithms [1, 47, 51] have been proposed to race detect programs that use futures, and the state-of-the art algorithm [1] provides an execution time bound of $O(T_1 + k^2)$, where k is the number of future operations. However, these prior algorithms must execute the program sequentially during race detection. The requirement of sequential execution is not just an implementation artifact but fundamental to how the algorithms maintain reachability. As the computation DAG unfolds dynamically during program execution, the reachability maintenance data structure must maintain reachability of all accesses occurred thus far. In these prior algorithms, the reachability data structure can only be maintained correctly assuming a particular traversal order of the computation DAG, which only the sequential execution guarantees. If the DAG folds in any other topological traversal order (which occurs during parallel executions), the reachability data structure proposed by these prior algorithms no longer work correctly.

[1]Henceforth, when we say a race, we mean a determinacy race unless noted otherwise.

In this work, we propose F-Order, the first know *parallel* race detection algorithm that race detects a program with futures while executing the program in parallel. Given a computation with T_1 work and T_∞ span, our race detection algorithm runs in time $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(k + \lg r \lg \hat{k}))$ on P processors, where k is the number of future operations, r is the maximum number of readers per memory location, and $\hat{k}$ is the maximum number of future operations done by a single future task, which is usually small.

To put this bound into perspective, a provably efficient parallel scheduler can execute a baseline program (i.e., no race detection) in time $O(T_1/P + T_\infty)$ [3]. Our race detection algorithm incurs additional $O(k^2)$ work, like the state-of-the-art sequential algorithm [1], but this additional work can be parallelized. It also incurs a multiplicative overhead of $\lg \hat{k}$ on the work term (which is small) and a multiplicative overhead of $(k + \lg r \lg \hat{k})$ on the span term, where k likely dominates the other term.

We have implemented and empirically evaluated a prototype system based on F-Order. The empirical results indicate the reachability component incurs little overhead and that the race detection obtains similar scalability as the baseline. Moreover, we have compared our parallel race detector against FutureRD, the state-of-the-art sequential race detector for futures [49, 51]. Empirical results indicate that, even though our parallel race detector incurs higher overhead on one-core execution, the fact that we can race detect while executing the program in parallel quickly pays off in absolute execution times.

Contributions

- We propose F-Order, the first known parallel race detection algorithm that race detects programs with futures. We provide the key intuitions behind our parallel algorithm (Section 4) and prove its correctness (Section 5).
- We show that F-Order can race detect a computation with work T_1 and span T_∞ in expected time $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(k + \lg r \lg \hat{k}))$ on P processors, where k is the number of future operations, r is the maximum number of readers per memory location, and $\hat{k}$ is the maximum number of future operations done by a future task (Section 6).
- We implemented a prototype of F-Order and empirically evaluate its scalability and overhead (Section 7). Experimental results indicate that F-Order scales well and the absolute running times of all benchmarks with race detection using F-Order on 4 cores or more beat the state-of-the-art sequential race detector [51].

2 Preliminaries

Language Constructs Considered. Fork-join parallelism can be expressed using two keywords: `spawn` and `sync`. When a function F ***spawns*** off another function G by prefixing the invocation with `spawn`, the execution of G may

operate in parallel with the continuation of F. The invocation of a `sync` specifies that all previously spawned functions must return before the control can pass `sync`.[2]

Future parallelism can be expressed using `create` and `get`. Similar to `spawn`, when a function F spawns off another function G by prefixing the invocation with `create`, the execution of G may operate in parallel with the continuation of F. We refer to this instance of G as a ***future task***. Unlike `spawn`, however, the invocation of `create` returns a ***future handle***, with which the execution of the future task G is associated. By invoking a `get` on the handle, the control cannot pass beyond `get` until the future task finishes and deposits its result into the handle. An invocation of a `sync` does not affect future tasks spawned off with `create`; `sync` can freely continue without waiting for future tasks to return.

Modeling Parallel Computations. One can model the execution of a parallel computation as a ***directed acyclic graph*** (or ***DAG***), in which a node represents a sequence of instructions without parallel control and an edge represents a control dependence between two nodes.

The execution of a `spawn` creates a ***spawn node*** with two outgoing edges: one to the spawned function (drawn as the left sub-DAG) and one to the continuation of the caller (drawn as the right sub-DAG). The execution of a `sync` creates a ***sync node*** with multiple incoming edges, one from the end of each spawned function to the sync node that represents the continuation after `sync`. Without loss of generality, we shall assume that a sync code consists of only two incoming edges — it is not difficult to convert a sync node with multiple incoming edges into a chain of sync nodes, each with two incoming edges.

The execution of a program that uses only fork-join parallelism generates a ***series-parallel (SP) DAG*** [52] with nice structural properties. Specifically, a SP DAG is a planar DAG with a unique ***source node*** with no incoming edges and a unique ***sink node*** with no outgoing edges. A SP DAG can be constructed recursively using series and parallel compositions where the spawn and sync nodes are well-nested.

The execution of a `create` generates a ***create node*** with two outgoing edges, one to the future task (drawn as the left sub-DAG) and one to the continuation of the caller (drawn as the right sub-DAG). The execution of a future task terminates with a ***put node***, which is the last node to execute in the future task that deposits the result into its future handle. The execution of a `get` terminates the current node v, and generates a ***join node*** with two incoming edges — one from v, referred to as join node's ***local parent***, and one from the put node of the corresponding future task, referred to as the join node's ***future parent***. We refer to the edge generated by `create` to the future task and the edge generated by `get` from the put node to the join node as ***non-SP edges*** as these edges do not conform to the well-nested structures that arise from series and parallel compositions for SP DAGs.

When a program uses both fork-join and future parallelism, the execution generates a ***nearly series-parallel (NSP) DAG*** $G_N = (D_{sp}, E_{non})$, a DAG formed by a set D_{sp} of SP DAGs connected by a set E_{non} of ***non-SP*** edges. Since each future task may contain fork-join parallelism, the execution of a future task can be modeled as its own SP DAG. If the future task executes a `create`, the spawned-off future task is a separate SP DAG, connected via non-SP edges.

Other Definitions. The following types of nodes are special: spawn, sync, create, join, and put nodes. All other nodes are ***common nodes***. Given two nodes u and v, we say u and v are ***in series*** if a directed path exists between u and v; otherwise they are ***in parallel***. If a path exists between u and v, we say u is an ***ancestor*** of v, and v is a ***descendent*** of u. Furthermore, we say create and put nodes are ***non-SP nodes***, which are special in that they have an outgoing non-SP edge. If there is a path from u to v, and u is either a create or put node, we say u is a ***non-SP ancestor*** of v.

Parallel Schedule and Performance Metrics. During execution, the DAG unfolds dynamically, and the scheduler maps execution of nodes onto processing cores, in a way that respects the dependences expressed by the parallel control constructs. A node v only becomes ready to execute when all its ancestors have finished executing. A valid parallel schedule is a topological sort of the nodes in the DAG.

There are two important metrics for measuring performance of a DAG: assuming each node takes unit time to execute, its ***work*** (denoted as T_1), the number of nodes in the DAG, or how long it takes to execute the DAG on one core; and its ***span*** (denoted as T_∞), the length of a longest path in the DAG, or how long it takes to execute the DAG on infinitely many cores. A DAG can be scheduled efficiently using a ***work-stealing*** scheduler [2, 3, 7, 8]. Given a DAG with T_1 work and T_∞ span, a work-stealing scheduler can execute the computation in time $T_1/P + O(T_\infty)$.

3 Related Work

Race Detection for Task Parallel Code. One can exploit the structural property of fork-join parallelism to obtain efficient race detection algorithms when the program uses only fork-join parallelism. Mellor-Crummey [29] is the first to show that, for a parallel race detector, it suffices to store two readers and a single writer per memory location in the access history. Feng and Leiserson [13, 14] provided the first provably efficient sequential race detection algorithm that is near-optimal. Bender et al. [4] proposed the first asymptotically optimal sequential algorithm and the first provably efficient parallel algorithm with a $O(P)$ overhead on the span term, where P is the number of cores used during execution. Finally, Utterback et al. [50] proposed the first asymptotically optimal parallel algorithm.

[2]Many task parallel platforms also support parallel loops, which can be thought of as syntactic sugar that compiles down to `spawn` and `sync`.

For race detecting pipeline parallelism, which is another task parallel paradigm with nice structural properties, Dimitrov et al. [12] proposed the first sequential algorithm that is near-optimal; Xu et al. [55] later proposed the first asymptotically optimal parallel algorithm.

Lee and Schardl [25] proposed a sequential race detection algorithm for fork-join computations with reductions, where the DAG is series-parallel except when executing reductions.

A few algorithms [1, 47, 51] have been proposed to race detect programs that use futures. Surendran and Sarkar [47] proposed the first algorithm to race detect a program that uses futures, but their algorithm runs sequentially and can incur large overhead, $O(T_1(f + 1)(k + 1)\alpha(m, n))$, where f is the number of future tasks, k is the number of future operations, m is the number of memory accesses, n is the number of parallel control constructs executed, and α is the functional inverse of Ackermann's function. Later, Agrawal et al. [1] improved the bound to $O(T_1+k^2)$, although the work is theoretical and no implementation of the algorithm exists. Finally, Utterback et al. [51] separated the use of futures into two classes — ***structured*** use of futures that imposes certain programming restrictions on where the future get can occur,[3] and ***general*** use of futures that does not impose such a restriction. By distinguishing the two, Utterback et al. [51] observed that programs that use structured futures can be race detected much more efficiently, in time $O(T_1\alpha(m, n))$. For general use of futures, Utterback et al. [51] proposed an algorithm (and its corresponding implementation) that executes in time $O((T_1 + k^2)\alpha(m, n))$.

Race Detection for Pthreaded Code. Much work has been done on race detecting pthreaded code. Like task parallel code, an execution of a pthreaded code can also be represented as a DAG and thus the problem of race detection boils down to checking for reachability of nodes in the DAG. Unlike task parallel code, however, the dependences in a pthreaded execution are formed via lock operations (as opposed to via high-level constructs) whose ordering forms ***happens-before (HB) relations***. Thus, the dependences are *schedule* dependent and nondeterministic for a given input. Thus, race detecting pthreaded code cannot be sound and complete for a given input as the number of possible schedules grows quickly.

The dependences formed via lock operations can be arbitrary with no structural properties. Researchers have proposed lock-set based algorithms [41, 53] that provide wide coverage but cannot precisely capture the dependences and may generate many false positives. Researchers have also proposed a Vector-clock (VC) based approach [16]) that precisely captures the dependences, but only for a given schedule. Hybrid approaches have also been explored [33, 36, 42, 56] that incorporate VC and lock-sets in order to achieve a compromise between coverage and precision. Finally, predictive analysis has been proposed as a method to explore alternative feasible schedules among nearby instructions in order to maintain precision while increasing the coverage [23, 26, 40, 44].

Even though a VC-based algorithm can capture arbitrary dependences and in principle can be applied to task-parallel code with futures, naively applying it to task-parallel code would be impractical. It requires storing a vector-clock of length n with each memory location, where n is the number of nodes in the computation DAG (which can be on the order of millions). Each memory access requires querying this vector, resulting a $O(n)$ overhead per access. In contrast, for task parallel code, one can exploit the structural properties in the DAG to encode reachability much more efficiently. In the case where futures are used, our algorithm still exploits the structural properties that exist within a single future task (which is an SP DAG) to allow for efficient reachability queries and handles the arbitrary dependences that arise due to non-SP edges in a special way. By doing so, our algorithm incurs an additional $O(k^2)$ space overhead across the entire execution and additional $O(\lg \hat{k})$ work per access, where k is the number of future operations, and $\hat{k}$ is the maximum number of future operations done by a single future task.

Race Detection for Event-Driven Applications. More recently, researchers have begun to study algorithms to race detect event-driven systems, such as web pages [35, 39] and mobile applications [5, 19, 20, 28]. A key challenge here, unlike the pthreaded code, is to establish a precise ***causality model***, i.e., how to capture happens-before relations between asynchronous events. These prior works mainly differ in their formulations of the causality model, which can be specific to the application domain (i.e., a model for web applications written in JavaScript will differ from that for Android applications). Similar to pthreaded code, once the causality model is established, an execution of an event-driven system can be represented as a DAG with arbitrary dependences, and the problem of race detection boils down to checking for reachability of nodes in the DAG.

Most prior work in this domain performs reachability queries by building and traversing the DAG explicitly [5, 20, 28, 35, 39], which can incur high overhead either during the DAG construction (i.e., the causality model requires traversing through the execution trace a few times to finalize the DAG) or during detection. Work by [39] builds the DAG but improves the overhead by running a VC-based algorithm on the DAG using "chain decomposition," where the width of a vector is determined by the number of chains in the DAG. Work by [5] further improves the overhead by proposing a causality model where the DAG can be constructed with almost a single pass of the execution trace. Finally, work by [19] proposes a causality model that allows

[3]The structured use of futures is first proposed by Herlihy and Liu [18] in the context of studying cache misses.

one to run a VC-based algorithm (with chain decomposition) with single-pass over the trace without building the DAG explicitly. This line of work differs from our work in that, the causality model (or how dependences form) for task parallel code with futures is much simpler, allowing us to encode reachability without explicitly building the DAG. While some work [5, 19, 39] proposed heuristics to optimize a VC-based algorithm, the optimizations are domain specific and not applicable in our case.

Use of Futures. Surendran and Sarkar [46] propose compiler analysis to automatically generate parallel code from serial code for pure functions using futures. Voss et al. [54] investigated methods to detect (and prevent) deadlock in applications that use futures. Finally, researchers have studied scheduling strategies and the corresponding performance bounds when a task parallel code uses futures [18, 43, 45].

4 Overview of F-Order

This section provides an overview of F-Order. The use of futures generates non-SP edges (defined in Section 2) that form arbitrary dependences and thus lack the structural properties that fork-join parallelism enjoys. This has two implications. First, it no longer suffices to store only a constant number of accessors per memory location in the access history. Second, the reachability can no longer be maintained efficiently by exploiting the structural properties. We discuss each of the components in turn, the intuitions behind F-Order, and how F-Order addresses the challenges.

4.1 Access History in F-Order

For fork-join parallelism, due to the nice structural properties of SP DAGs, it is sufficient to store only the "left-most" and "right-most" readers per memory location during parallel execution [29]. One can prove that a reader omitted by the access history can race with a writer if and only if the writer also races with either the left-most or the right-most reader. Thus, we do not miss a race by omitting such a reader in the access history. For a program that uses futures, however, we no longer have the same structural properties — there are no clear "left-most" and "right-most" readers that one can store to subsume potential races with other readers. Thus we must store all readers encountered until a sequential writer comes along.

How F-Order maintains access history follows the same strategy as the prior state-of-the-art sequential algorithm [1]. For each memory location l, F-Order stores a ***last writer***, `last-writer`(l), which is the last node that wrote to l, and a list of readers `reader-list`(l) that read l since `last-writer`(l). Whenever a node r tries to read a memory location l, F-Order performs a reachability query between r and `last-writer`(l) to see if r races with the last writer. If so, a race is reported. If not, r is added to `reader-list`(l). Whenever a node w writes to a memory location l, F-Order checks w against all the readers in `reader-list`(l) and the last writer `last-writer`(l). If any of the `reader-list`(l) or `last-writer`(l) is logically in parallel with w, a race is reported. Otherwise, F-Order empties the `reader-list`(l) and sets `last-writer`(l) to be w. As argued by Agrawal et al. [1], we won't miss any races by emptying `reader-list`(l) because any future access that races with a node in `reader-list`(l) must also race with w. The fact that the prior algorithm executes the program sequentially and F-Order executes it in parallel does not change the correctness argument, so long as F-Order synchronizes the access history data structure correctly.

Agrawal et al. [1] also show that that the total number of reachability queries per reader is bounded by two. F-Order provides the same bound on the number of reachability queries per reader. Since F-Order executes in parallel, however, we must also consider how the reachability queries impact the span, which we discuss in Section 6.

4.2 Reachability Maintenance in F-Order

The Challenges. A computation that uses futures can be modeled as a nearly-series-parallel DAG (NSP DAG), consisting of a set of SP DAGs connected via non-SP edges. Given two nodes, if they are connected by only SP edges, one can perform a reachability query on them efficiently by applying the reachability maintenance algorithm used for fork-join parallelism from prior work [4, 15, 50]. The challenge is to handle the reachability queries efficiently when the two nodes are possibly connected in part by non-SP edges.

The state-of-the-art prior algorithm [1] encodes the reachability that arises due to non-SP edges explicitly using an auxiliary graph $\mathcal{R}$. Unfortunately, the maintenance of $\mathcal{R}$ heavily depends on traversing the computation DAG in a left-to-right depth-first fashion (i.e., executing the DAG sequentially). Thus, this prior algorithm simply cannot be parallelized, since a parallel execution can traverse the DAG in any order (as long as it is a topological sort of the DAG), which breaks the invariants required by $\mathcal{R}$ to keep track of reachability correctly. Thus, the reachability maintenance in F-Order has to use an entirely different strategy.

The Intuitions. Based on how we model the computation, each future task forms its own SP DAG, and different SP DAGs can only be connected via non-SP edges. That means, if two nodes are in series and connected by only SP edges then they must belong to the same SP DAG, and one can utilize a prior parallel algorithm such as WSP-Order [50] to correctly answer reachability queries between them. WSP-Order cannot encode the reachability of two nodes correctly if they are connected via non-SP edges, regardless of whether they belong to the same SP DAGs or not. Thus, we need some other means to encode reachability between two nodes if they are connected via non-SP edges.

The key observation is as follows. Given two nodes u and v connected via non-SP edges, some node w must exist in the path between u and v, where w is a non-SP ancestor of v that is in the same SP DAG as u. That is, the prefix of the path (from u to w) contains only SP edges, and the suffix (from w to v) containing at least one non-SP edge (outgoing from w). Thus, given two nodes u and v possibly connected in part by non-SP edges, we can query their reachability efficiently if we can quickly determine if such an ancestor w exists, who is 1) an non-SP ancestor of v in the same SP DAG as u and 2) in series with u via only SP edges.

The FOM Data Structure. To quickly determine whether such an non-SP ancestor w of v exists, F-Order employs an enabling data structure called the ***Future Order-Maintenance*** (or ***FOM*** for short) data structure per node in the NSP DAG. An FOM data structure for v stores all v's non-SP ancestors, organized into groups, where each ***group*** contains non-SP ancestors from the same SP DAG.

Upon execution of a node v, its FOM data structure will be complete and its content fixed because v's FOM data structure contains only v's non-SP ancestors, and these must have already been discovered by the time v executes as the scheduler guarantees that a node cannot execute until all its ancestors have executed. If v accesses some memory location that some node u accessed previously, F-Order queries v's FOM data structure to find the group g that holds the non-SP ancestors from the same SP DAG as u. F-Order then checks with g to determine whether some w exists that is reachable from u. If a node is in the group g, by definition it is in the same SP DAG as u. Then, we simply need to check if w is in series with u.

A naive implementation would be to check reachability against every node in g, taking time linear in the size of g. Ideally, we would like to quickly eliminate non-viable candidates in the group and avoid querying every single node in the group. The key insight of F-Order involves identifying the correct auxiliary data to store with each non-SP ancestor in a group so that the process of elimination can be done quickly.

It turns out that, to perform the process of elimination within a group, we simply need to organize nodes in a given group as follows. First, store the nodes in a total order, called the ***English*** order [32], that corresponds to the depth-first-left-to-right traversal of nodes in the corresponding SP DAG. The English order is well defined among nodes in the same group, because a group contains only nodes from the same SP DAG with no non-SP edges. Second, with each node w in the group, additionally store w's ***furthest descendent*** that is also within the same group; that is, some node z that is reachable from w that is also in the group such that no other node y in the group is reachable from z (formally defined in Section 5). We will elaborate on the detailed construction of FOM data structures and discuss how F-Order uses them to perform reachability queries in Section 5.

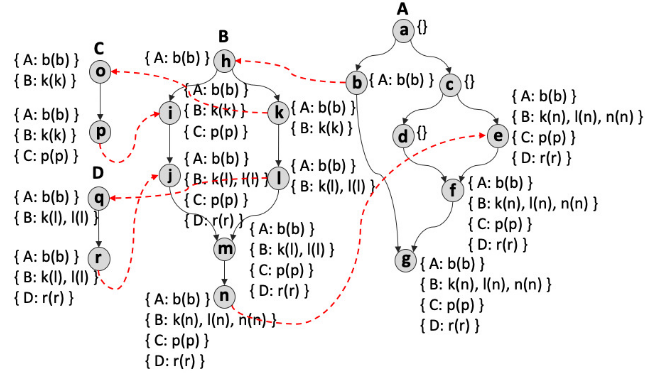

Figure 1. An example of a NSP DAG with every node's FOM data structure shown. In this NSP DAG, four SP DAGs exist, ID'ed as A, B, C, and D, with A being the main SP DAG and the others being the spawned future tasks. The non-SP edges are shown as thick dashed edges. Each node has its own instance of FOM data structure, containing entries of $\{key : value\}$ pairs, where the key is the ID of an SP DAG and the corresponding $value$ is a set of non-SP ancestors from the SP DAG. The parentheses next to each non-SP ancestor shows its furthest descendant in the group.

4.3 An Illustrating Example

Figure 1 shows the static snapshot of an NSP DAG with all its FOM data structures shown. Note that the parallel execution unfolds the NSP DAG dynamically, revealing each node as it becomes ready (i.e., all its ancestors have executed). Nevertheless, as discussed earlier, the content of an FOM data structure for node v is fixed by the time v executes, so this dynamic unfolding of the DAG does not change the content of the FOM data structures shown.

For now, we shall focus solely on the organization of an FOM data structure and how we use it to perform reachability queries. Take node f: it contains multiple non-SP ancestors. They are grouped into four entries in f's FOM data structure, as they respectively belong to four different SP DAGs. In particular, f has nodes k, l, and n as its non-SP ancestors, all from SP DAG B. Thus, its FOM data structure contains an entry with group keyed by B, and the corresponding value is a list of non-SP ancestors ordered in their English order. (Note that the nodes in a given SP DAG are labeled alphabetically according to their ordering in the English order.)

Say f is being executed, and F-Order wants to check if f is reachable from node i. Since i belongs to SP DAG B, F-Order checks f's FOM data structure for the group indexed with B, which returns the list k, l, and n (with every node having n as its furthest descendant). Since n is reachable from i, F-Order concludes that f is reachable from i. In this case, the group for B is small, containing only three nodes. However, the size of a group can be larger, and ideally we want F-Order to quickly home in on n and not check i against every node in the group. This is where the English ordering and the

auxiliary data of furthest descendants become useful, which we discuss in Section 5.

5 Details of F-Order and Its Correctness

This section presents the full detail of F-Order and its correctness proof. F-Order consists of two parts: a ***construction*** algorithm that builds and maintains the FOM data structure for each node and a ***reachability-query*** algorithm that checks whether a given pair of nodes are reachable from one another. We discuss each in turn. Throughout the section, we shall refer back to Figure 1 as an illustrating example.

Notations. Given an NSP DAG $G_N = (D_{sp}, E_{non})$, which consists a set of SP DAGs connected via non-SP edges, we assume each SP DAG $d \in G_N$ is assigned with a unique integer identifier, denoted as $SP(d)$. Given a node u, we overload the notation and use $SP(u)$ to denote the ID of the SP DAG containing u. Given two nodes u and v, we use $u \rightsquigarrow v$ to denote the presence of a directed path from u to v. We use $u \rightsquigarrow_{sp} v$ if the path comprises only SP edges and $u \rightsquigarrow_{nsp} v$ if the path comprises any non-SP edge. We say $u \prec v$ iff $u \rightsquigarrow v$, and $u \preceq v$ iff either $u \prec v$ or $u = v$. If u and v are in the same SP DAG $d \in D_{sp}$, we say $u \prec_d v$ iff $u \rightsquigarrow_{sp} v$; we say $u \parallel^d_{left} v$ iff node u is ***left of*** v in SP DAG d, where u is in the left sub-DAG of d and v is in the right sub-DAG of d. Both notations $\prec_d$ and $\parallel^d_{left}$ are only applicable to nodes in the same SP DAG d. They specify the English order in that, if u is before v in the order, then either $u \prec_d v$ or $u \parallel^d_{left} v$. For instance, in Figure 1, $c \prec_d g$ and $b \parallel^d_{left} e$. On the other hand, h and g cannot be related using these operators.

5.1 Construction of FOM Data Structures

As the NSP DAG unfolds, nodes become ready and get executed. When a node v executes, F-Order constructs an FOM data structure for v, denoted as $v.fom$, whose content is complete at the beginning of v's execution to allow for reachability queries for memory accesses performed by v. An FOM data structure is organized as a hash table, hashing $SP(d)$ to its corresponding group that stores all of v's non-SP ancestors that belong to the SP DAG d.

For a given group g, an element e in g has two fields: $e.node$ and $e.desc$. Field $e.node$ stores the actual non-SP ancestor. Field $e.desc$ stores the ***furthest descendent*** of $e.node$ in g. We say a node v is the furthest descendent of $e.node$ in g iff (1) $e.node \preceq_d v$, (2) there exists an element x in g such that $x.node = v$, (3) for any other element y in g, we have $v \not\prec_d y.node$. Intuitively, the furthest descendant of $e.node$ is another non-SP ancestor of v stored in the same group g that is a descendant of $e.node$ and $e.node$ has no other descendant in the group that is further out.

Properties of an FOM data structure. An FOM data structure maintains the following properties.

1. For a non-SP node w such that $w \preceq v$, there exists a group containing element x in $v.fom$ such that $x.node = w$.
2. Given two elements x and y in group g, $x.node$ and $y.node$ are in the same SP DAG d. If $x.node \prec_d y.node$ or $x.node \parallel^d_{left} y.node$, then x is before y in g.
3. Given an element x in group g, its furthest descendant field is maintained properly. That is, (1) $x.node \preceq_d x.desc$, (2) there exists an element y in g such that $y.node = x.desc$, (3) for any other element z in g, we have $x.desc \not\prec_d z.node$.

Property 1 states that, given a node v in the NSP DAG, $v.fom$ has all v's non-SP ancestors. Property 2 states that the elements in a group are stored in the English order. Since a group stores only nodes from the same SP DAG, their relationships ($\prec_d$ or $\parallel^d_{left}$) can be maintained and queried efficiently using the prior parallel algorithm WSP-Order [50] designed for fork-join parallelism. Property 3 states that the furthest descendents for every element is maintained correctly.

The construction algorithm in F-Order assumes the following helper functions that operate on FOM data structures.

- `FOM-Insert`(fom, v): Given an instance fom and a non-SP node v, `FOM-Insert` returns a new instance of FOM created by copying over the content of fom and inserting v into the appropriate group stored in fom.
- `FOM-Merge`(fom_1, fom_2): Given two instances fom_1 and fom_2 of FOM, `FOM-Merge` returns a new instance of FOM created by merging the contents of fom_1 and fom_2.

Algorithm 1: F-Order, Construction

```
1  Function CommonOrSpawn(v)
2  |  v.fom = v.parent.fom
3  Function Sync(v)
4  |  let u be the corresponding spawn node of v
5  |  if u.fom ≠ v.lparent.fom and
   |    u.fom ≠ v.rparent.fom then
6  |  |  v = FOM-Merge(v.lparent.fom, v.rparent.fom)
7  |  else if u.fom ≠ v.lparent.fom then
8  |  |  v.fom = v.lparent.fom
9  |  else
10 |  |  v.fom = v.rparent.fom
11 Function CreateOrPut(v)
12 |  v.fom = FOM-Insert(v.parent.fom, v)
13 Function Join(v)
14 |  v.fom = FOM-Merge(v.local.fom, v.future.fom)
```

Algorithm 1 shows the pseudocode of the construction algorithm. F-Order constructs FOMs for all nodes by propagating the presence of non-SP nodes (i.e., create or put) to all its descendents during parallel execution. The algorithm initializes an empty instance of FOM for the source node of the main DAG since it doesn't have any ancestor. Subsequently, it executes nodes of the DAG in any valid

order: a node can be executed when all it's ancestors have finished executing. As each node v executes, the algorithm calls different functions based on v's node type.

If v is a common or spawn node, v has only one parent $v.parent$. Any non-SP ancestor of $v.parent$ (possibly including $v.parent$ itself) is also a non-SP ancestor of v. Thus, the algorithm sets $v.fom = v.parent.fom$ (line 2).

If v is a create or put node, besides inheriting all its parent's non-SP ancestors, v also needs to insert itself into its own FOM data structure. Therefore, the algorithm invokes `FOM-Insert` to insert itself into $v.parent.fom$ (line 12), which implicitly creates a new instance of FOM that copies the content from $v.parent.fom$ and inserts itself into the appropriate group.

If v is a join node, then v has two parents: a local parent $v.local$ and a future parent $v.future$. Then the algorithm creates a new instance of FOM by merging $v.local.fom$ and $v.future.fom$ (line 14).

Finally, if v is a sync node, then v has two parents: a left parent $v.lparent$ and a right parent $v.rparent$. However, it is not always necessary to merge $v.lparent.fom$ and $v.rparent.fom$. By the structural properties of an SP DAG, we know that a sync node v has a corresponding spawn node u and two SP sub-DAGs in between: the left sub-DAG G_L and the right sub-DAG G_R. The source node of G_L inherits $u.fom$, which can only change when G_L contains either a create node or a join node (i.e., with an incoming put edge). If G_L does not contain either create or join node, $v.lparent.fom$ remains the same as $u.fom$. Similarly, the same thing holds for G_R and $v.rparent.fom$. The merge is only necessary when both $v.lparent.fom$ and $v.lparent.fom$ have changed compared to $u.fom$. Thus, the algorithm checks for whether both have changed, and if so calls `FOM-Merge` (line 6). Otherwise, if only one changed, $v.fom$ inherits the one that has changed (lines 8 and 10). If neither has changed, it doesn't matter which one $v.fom$ inherits from. Take node f in Figure 1 for instance: its corresponding spawn node is c and only the right sub-DAG (i.e., e) has its FOM data structure changed from c. Thus, f simply inherits its FOM data structure from its right parent e. The node g, on the other hand, constructs its FOM data structure by merging the FOM data structures from both of its parents.

For performance reasons, it is important that F-Order calls `FOM-Merge` only when both sub-DAGs execute nodes that cause their respective FOM data structure to change, a fact that we use when proving the performance bound of F-Order in Section 6.

Lemma 5.1 states that Property 1 always holds for an FOM data structure:

Lemma 5.1. *Given a node v, F-Order constructs a FOM instance $v.fom$ that stores all the non-SP ancestors (i.e., future create and put nodes) of v (**Property 1**).*

Proof Sketch. One can show this inductively by the nodes executed during parallel execution: provided that the FOM instance(s) of v's parent(s) satisfy Property 1, the construction of v's FOM also satisfies Property 1. □

To show that the construction algorithm satisfies the Properties 2 and 3, we need to examine the `FOM-Insert` and `FOM-Merge` in more detail. Due to space constraints, we discuss what these functions do at a high-level and omit the full pseudocode.

Insert Operation. At `FOM-Insert`(fom, v), we create a new FOM fom_{new} by copying the content of fom into fom_{new}. Then we check if fom_{new} contains a group g with $SP(v)$. If so, we call `Group-Insert`(g, v), which returns a new group g_{new} with a copy of g's content and v added, and we replace g with g_{new} in fom_{new}. If not, we simply create a new empty group g_{new} with v added, and add g_{new} to fom_{new}.

Algorithm 2: Helper function: Group-Insert

```
15 Function Group-Insert(g, v)
16     g_new = new Group() //create a new group
17     i = j = 1
18     while j ≤ g.length do
19         x = g[j++] //the jth group element in g
20         if x.node ≺_d v ∨ x.node ||^d_left v then
21             // copy constructor copying the content of x into y
22             y = new Group-Element(x)
23             // update furthest descendant if necessary
24             if x.desc ≺_d v then y.desc = v
25             g_new[i++]=y // insert group element y into g_new
26         else break // found the right position for v
27     g_new[i].node = g_new[i].desc = v
28     i = i + 1
29     //copy the remaining elements of g into g_new
30     while j ≤ g.length do
31         g_new[i++] = new Group-Element(g[j++])
32     return g_new
```

Algorithm 2 shows the helper function `Group-Insert`, which uses linear search to find the correct position of v in g_{new}, which keeps Property 2. Furthermore, for a node u in g_{new} such that $u \prec_d v$, v checks its furthest descendent to see if it should be replaced with v; if so, update it in g_{new} (line 24). For any u' that is positioned after v in g_{new}, v cannot be u''s furthest descendant as either $v \prec_d u'$ or $v \parallel^d_{left} u'$ and thus we simply copy over the rest (line 31).

We can conclude the following lemma for `FOM-Insert`:

Lemma 5.2. *Given a FOM fom that satisfies Properties 2 and 3,* `FOM-Insert`(fom,v) *returns a new FOM with the content of fom and v inserted that satisfies Properties 2 and 3.*

Merge Operation. `FOM-Merge` is used in the construction algorithm to merge two FOM instances. At `FOM-Merge`(fom_1, fom_2), we create a new FOM fom_{new}.

We first iterate through groups in fom_1 and insert them into fom_{new}. We then iterate through groups in fom_2. For each group g_2 in fom_2, we check if some group g_1 with $SP(g_2)$ already exists fom_{new}. If so, we call `Group-Merge(`g_1, g_2`)`, which returns a new group g with merged content, and we replace g_1 with g in fom_{new}. If not, we simply insert g_2 into fom_{new}.

Algorithm 3: Helper function: Group-Merge

```
33 Function Group-Merge(g1, g2)
34     g_new = new Group() //create a new group
35     i = j = k = 1
36     while i ≤ g1.length ∧ j ≤ g2.length do
37         x = g1[i] //the ith group element in g1
38         y = g2[j] //the jth group element in g2
39         if x.node ≺_d y.node ∨ x.node ||^d_left y.node then
40             z = new Group-Element(x)
41             i = i + 1
42         else if x.node = y.node then
43             z.node = x.node
44             if x.desc ⪯_d y.desc then z.desc = y.desc
45             else z.desc = x.desc
46             i = i + 1; j = j + 1
47         else
48             z = new Group-Element(y)
49             j = j + 1
50         g_new[k++] = z //insert group element z into g_new
51     //copy the remaining elements of g1 or g2 into g_new
52     while i ≤ g1.length do
53         x = g1[i++]
54         g_new[k++] = new Group-Element(x)
55     while j ≤ g2.length do
56         y = g2[i++]
57         g_new[k++] = new Group-Element(y)
58     return g_new
```

Algorithm 3 shows `Group-Merge`, which merges two groups while maintaining the English order during merge, and its operation is akin to the merge step in merge sort. Using a similar correctness proof as merge sort, we can conclude the following lemma:

Lemma 5.3. *Given fom_1 and fom_2 that satisfy Property 2,* `FOM-Merge(`fom_1, fom_2`)` *returns a new FOM instance with the merged content of fom_1 and fom_2 that satisfies Property 2.*

What is not obvious is that Property 3 is also maintained by `Group-Merge`. Consider the process of merging two groups g_1 and g_2. Given an element x in g_1, $x.desc$ stores the furthest descendent of $x.node$ in the scope of g_1. However, it is possible that there exists a node u in g_2 such that $x.desc \prec_d u$, which means that u should become the new $x.desc$ after merging g_1 and g_2 into g_{new}. It may seem that `Group-Merge` needs to check $x.desc$ against every single node in g_2. It turns out that it is sufficient to only check $x.desc$ against $y.desc$ of an element y in g_2 such that $x.node = y.node$, and during the merge we are guarantee to compare x against y for $x.node = y.node$ (lines 42–46). Lemma 5.4 states that doing so is sufficient to maintain Property 3.

Lemma 5.4. *Given two groups g_1 and g_2 that satisfy Property 3,* `Group-Merge` *merges the content of g_1 and g_2 into g_{new} while maintaining Property 3 for g_{new}.*

Proof. Given an element x in g_1, say there exists a node u stored in g_2 such that u is the new furthest descendent of $x.node$. Then, we have $x.node \prec_d u$. Also, suppose g_2 is maintained by $v.fom$. Then we have $u \preceq v$ since all the nodes stored in $v.fom$ are v's ancestors, which leads to $x.node \prec v$. By Property 1, we know $v.fom$ stores all of v's non-SP ancestors. Thus, $x.node$ must be stored in some group of $v.fom$. Since $x.node$ and u are in the same SP DAG, we are guaranteed that $x.node$ is also in g_2. Thus, there must exist an element y in g_2 such that $y.node = x.node$ and $y.desc = u$. Thus, similar to the merge step in merge sort, there must exist an iteration that performs the comparison between x and y. As a result, it is sufficient to check for update for $x.desc$ against $y.desc$ in such an iteration. □

5.2 Reachability Queries Using FOM

We now describe how we do the reachability query between two nodes u and v. Recall the intuitions discussed in Section 4.2. The following lemma formalizes this intuition:

Lemma 5.5. *Give two nodes u and v in G_N, we have $u \prec v$ iff one of the following is true: 1) $u \prec_d v$; or 2) $u \preceq_d w$, $w \prec v$ where w is a non-SP node (i.e., a create or put node).*

Proof. It is clear that if $u \prec_d v$, or $u \prec_d w$ and $w \prec v$, we have $u \prec v$. Now we show the other direction also holds. If $u \prec v$, there are two possibilities: 1) $u \rightsquigarrow_{sp} v$ or 2) $u \rightsquigarrow_{nsp} v$. The first case $u \rightsquigarrow_{sp} v$ is easy to see that u and v must be in the same SP DAG and thus we have $u \prec_d v$. Let's consider the second case $u \rightsquigarrow_{nsp} v$. Then the path must pass through some create or put node because all outgoing non-SP edges are incident on either create or put nodes. Now we prove that there exists a node w that $u \preceq_d w$. If u is a create or put node, then $w = u$. If not, u must have an outgoing SP edge. Therefore, we can break the non-SP path from u to v into a SP path and a non-SP path connected via some create or put node w. As a result, we have $u \rightsquigarrow_{sp} w$, i.e., $u \prec_d w$. □

Lemma 5.5 states that in an NSP DAG G_N, node u has a path to v iff: (1) u and v are in the same SP DAG d and there is an SP path between u and v in d; or (2) the path passes through a create or put node that breaks the path into an SP path and a non-SP path. The first case can be queried efficiently using prior work [50]. The latter case is where we apply the FOM data structures. Specifically, when querying for reachability between u and v, F-Order first queries if $u \prec_d v$ if they are in the same SP DAG; otherwise, F-Order

searches whether there exists a non-SP ancestor w stored with $v.fom$ such that $u \preceq_d w$.

Algorithm 4: Group-Search in Reachability Query

```
59 Function Precedes(u, v)
60   if SP(u) = SP(v) ∧ u ≺_d v then return TRUE
61   else
62     g = v.fom.find(SP(u))
63     if g then return Group-Search(u, g)
64     else return FALSE
65 Function Group-Search(u, g)
66   low = 1; high = g.length
67   while low ≤ high do
68     mid = (low + high)/2; m = g[mid]
69     if u ⪯_d m.node then
70       return TRUE
71     else if m.node ≺_d u ∨ m.node ||^d_left u then
72       low = mid + 1
73     else // must be u ||^d_left m.node
74       if u ≺_d m.desc then return TRUE
75       else high = mid − 1
76   return FALSE
```

Algorithm 4 shows the pseudocode for Precedes(u,v), which checks if $SP(u) = SP(v)$ and $u \prec_d v$. If so, $u \prec v$ and we are done. If not, we then check if a non-SP path exists using v's FOM data structure $v.fom$. We search for a group g with $SP(u)$ in $v.fom$; if found, we invoke Group-Search(u, g) to see if a w exists such that $u \preceq_d w$. If one is found, then $u \rightsquigarrow_{nsp} v$; if not, we conclude that no path exists between u and v.

By Property 2, elements in a group g is stored in a total English order. Group-Search uses this fact to apply a process of elimination akin to that in binary search. As hinted before, the process of elimination involves some complications — upon encountering an element x, in some cases, we must compare u against $x.desc$ (line 74) to correctly eliminate half of the remaining elements to check. This leverages Property 3 to guarantee correctness, which we discuss in Lemma 5.6.

Lemma 5.6. *Given a node u and a group g, the search in* Group-Search(u,g) *returns true iff there exists an element x in g such that $u \preceq_d x.node$; it returns false otherwise.*

Proof. First we show that if Group-Search(u,g) returns true, there exists an element x in g such that $u \preceq_d x.node$. This is evident from the code: Group-Search(u,g) only returns true when such a node is found (lines 70 and 74).

Now we show the other direction also hold: if an element x exists in g such that $u \preceq_d x.node$, Group-Search(u,g) returns true. That is, if such x exists, Group-Search(u,g) will find it by correctly eliminating half of the remaining elements that we don't need. We will examine this by cases on the if conditions executed in Group-Search(u,g).

By Property 2, if $x.node \prec_d y.node$ or $x.node \parallel^d_{left} y.node$, then x is positioned before y in g. Let's suppose that the element we are looking for is x and it exists. If $g[mid].node \prec_d u$ (first part of condition in line 71), then obviously $g[mid].node \prec_d x.node$ and we need to only search the array elements positioned after $g[mid]$. The code correctly performs the elimination (line 72).

Now we show if $g[mid].node \parallel^d_{left} u$ (second part of condition in line 71), then we have either $g[mid].node \prec_d x.node$ or $g[mid].node \parallel^d_{left} x.node$. Consider the left sub-DAG G_L containing $g[mid].node$ and the corresponding right sub-DAG G_R containing u. Say G is the SP sub-DAG consists of G_R and G_L. Then there are two possibilities for where $x.node$ can be: either $x.node$ is in G_R, then $g[mid].node \parallel^d_{left} x.node$, or $sink(G)$ (the sink node of G) $\preceq_d x.node$, then $g[mid].node \prec_d x.node$. In either case, $g[mid]$ must be positioned before x in g, and the code correctly performs the elimination (line 72).

We now consider the case that $u \parallel^d_{left} g[mid].node$ (line 73). Again, consider the SP sub-DAG G with the left and right sub-DAGs G_L and G_R. Say u is in G_L and $g[mid].node$ is in G_R. Then it could be either $sink(G) \preceq_d x.node$ or $x.node \in G_L$. In the first case, $x.node$ is also a descendent of $g[mid].node$. Recall that by Property 3, $g[mid].desc$ stores the furthest descendent node of $g[mid].node$. If $g[mid].node \prec_d x.node$, we are guaranteed that $sink(G) \preceq_d g[mid].desc$. Otherwise, we have $g[mid].desc \prec_d sink(G)$ and as a result $g[mid].desc \prec_d x.node$, which contradicts that $g[mid].desc$ is $g[mid].node$'s furthest descendent. Thus, if $sink(G) \preceq_d x.node$ is true, we must have $sink(G) \preceq_d g[mid].desc$, which leads to $u \prec_d g[mid].desc$ (line 74). Now we consider the second case, $x.node \in G_L$. In this case, we have obviously $x.node \parallel^d_{left} g[mid].node$. By Property 2, we can conclude that target x must be between positions low and $mid - 1$, and the code correctly performs the elimination (line 75). □

The last part of the proof in Lemma 5.6 makes it clear why we must store the furthest descendent with every element — even if the target node is in the part of the array that we eliminate, we are guaranteed to find it as another element's furthest descendent field. Take nodes i and f in Figure 1 for instance. Say f executes second, and we want to check if f is reachable from i. We find the group g with $SP(i) = B$ and invoke Group-Search(i, g). Even though Group-Search eliminates the second half of g, it still concludes that f is reachable from i (i.e., returns true), as we have n stored as $l.desc$ and $i \prec_d n$.

By Lemmas 5.5 and 5.6 and by the operations of Precedes, we can show the following theorem.

Lemma 5.7. *Provided that $v.fom$ satisfies Properties 1–3,* Precedes(u,v) *correctly returns true iff $u \rightsquigarrow v$ in G_N.*

Theorem 5.8. *Given two executed nodes u and v in NSP DAG G_N. F-Order correctly answers the reachability query between u and v.*

Proof. By Lemmas 5.1, 5.2, 5.3, and 5.4, one can inductively show that Properties 1–3 on any FOM is maintained at all times. Then by Lemma 5.7, F-Order can answer the reachability query correctly provided that the FOM instance of each executed node satisfies Properties 1–3. □

6 The Performance Bound of F-Order

This section proves the performance bound of F-Order. To perform a reachability query between two nodes in the same SP DAG, we utilize the parallel algorithm WSP-Order from Utterback et al. [50], including scheduling support for maintaining concurrent order-maintenance (OM) data structures. In our work, we augmented our scheduler similarly to provide support for such concurrent OM data structures.

To bound the overhead of WSP-Order and the use of concurrent OM data structures, we invoke the following lemma shown by Utterback et al. [50]:

Lemma 6.1. *Given an SP DAG with work T_1 and span T_∞, one can perform reachability maintenance and queries on the SP DAG in time $O(T_1/P + T_\infty)$ on P processors.*

To complete the time bound for F-Order, we need to account for the additional overhead incurred by the maintenance of and queries on the FOM data structures.

Lemma 6.2. *Given an NSP DAG with k number of future operations, the total number of* `FOM-Insert` *invocations is at most $O(k)$, each with $O(k)$ work.*

Proof. Each FOM data structure stores only non-SP ancestors, i.e., ancestors that are either create or put nodes. Since there are k future operations, each FOM instance can have at most $O(k)$ elements. Therefore, it takes $O(k)$ time to perform a single `FOM-Insert` operation (which creates a new copy so linear work is required). Moreover, by Algorithm 1, F-Order performs `FOM-Insert` only on create and put nodes, and thus `FOM-Insert` is invoked at most $O(k)$ times. □

Lemma 6.3. *Given an NSP DAG with k number of future operations, the total number of* `FOM-Merge` *invocations is at most $O(k)$, each with $O(k)$ work.*

Proof. `FOM-Merge` operates on inputs of size at most $O(k)$, and thus each operation incurs at most $O(k)$ work (like the merge operation in merge sort). By Algorithm 1, F-Order performs `FOM-Merge` only on join and sync nodes. Since there are at most k future operations, there are at most k join nodes. What's not as obvious is bounding the number of `FOM-Merge` invocations due to sync nodes.

By Algorithm 1, F-Order performs `FOM-Insert` on a sync only when the FOM data structures from both of its parents have changed from that of its corresponding spawn node.

Consider an SP DAG constructed recursively using n parallel compositions. We will call the outer-most SP DAG a level-0 DAG, or G^0, and call its left and right sub-DAGs level-1 DAGs, G^0_L and G^0_R, or simply G^1. Since this DAG is constructed with n parallel compositions, there are n levels of nested SP DAGs with n sync nodes, one for each level. Without loss of generality, we will show that, by adding a new incoming or outgoing non-SP edge into this DAG, the addition incurs at most one extra `FOM-Merge` on the closest enclosing sync node, but not on the other sync nodes at the outer level.

Imagine today we add an outgoing non-SP edge to the left sub-DAG at level i, G^i_L. Consider both the sync node s that joins together G^i_L and G^i_R and its corresponding spawn node f. The FOM instance from s's left parent would change. This may or may not prompt a `FOM-Merge` at s.

Case 1: Let's consider the case where it did. Then, it must be that there is also an incoming or outgoing non-SP edge in G^i_R, causing the FOM instance from s' right parent to change from that of the $f.fom$. If so, one extra `FOM-Merge` would be incurred compared to not adding that non-SP edge. From the perspective of the sync node s' at level $i-1$, however, this change does not affect whether s' performs a `FOM-Merge` or not. Without loss of generality, say the DAG consist of G^i_L and G^i_R is the left sub DAG at level $i-1$ (i.e., G^{i-1}_L). Without adding a new non-SP edge to G^{i-1}_R, s' will simply inherits the results of `FOM-Merge` at s.

Case 2: Let's consider the other case where this addition did not prompt s at level i to perform `FOM-Merge`. Then, it must be that, the FOM from s's right parent is the same as $f.fom$, the FOM from the corresponding spawn node. Then, s would have simply inherit the FOM from G^i_L, incurring zero additional `FOM-Merge` operations at level i. Now, it may incur an extra `FOM-Merge` at the sync node at level j for some $j < i$. But the same argument from case 1 can be applied to level j and the `FOM-Merge` stops at level j and not further.

Thus, we can conclude that there are at most $O(k)$ total `FOM-Merge` operations, each with $O(k)$ work. □

Now we put the overhead due to maintaining FOM data structures together.

Lemma 6.4. *Given an NSP DAG with work T_1 and span T_∞. F-Order runs in time $O((T_1 + k^2)/P + T_\infty k)$ on P processors to construct the reachability data structure, where k is the number of future operations.*

Proof. By Lemmas 6.2 and 6.3, the construction of FOM data structures incurs at most $O(k^2)$ work and in the worst case, $O(k)$ multiplicative overhead on the span (if F-Order performs a `FOM-Merge` or `FOM-Insert` on every single node along the span). Adding these overheads and applying Lemma 6.1, we obtain the bound. □

The bound shown in Lemma 6.4 accounts for only the construction overhead. To show the full performance bound, we must also account for the query overhead.

Lemma 6.5. *Given two nodes u and v in an NSP DAG G_N, a single reachability query* `Precedes(u, v)` *runs in time $O(\lg \hat{k})$,*

where $\hat{k}$ is the number of non-SP ancestors of v from the SP DAG containing u.

Proof. In the worst case, there is no direct SP path between u and v, and F-Order needs to perform a search to check if $v.fom$ stores any descendent node of u. Identifying a group g with $SP(u)$ in $v.fom$ takes constant time; if g exists, invoking `Group-Search(u, g)` takes at most $O(\lg \hat{k})$ time (akin to binary search). □

Theorem 6.6. *Given an NSP DAG G_N with work T_1 and span T_∞, F-Order can race detect G_N in parallel in time $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(k + \lg r \lg \hat{k}))$ on P processors, where k is the number of future operations, r is the maximum number of readers for a single memory location, and $\hat{k}$ is the maximum number of non-SP nodes in the same SP DAG.*

Proof. The total overhead incurred due to reachability queries is related to how the access history is managed. As discussed in Section 4, the number of readers per memory location at a given moment can be large. However, one can still show that, the total number of reachability queries throughout the execution is bounded by 2× the number of reads during execution [1]. Each query itself incurs $O(\lg \hat{k})$ overhead Lemma 6.5. Thus, given an NSP DAG with work T_1 and span T_∞, the total work incurred by reachability queries using F-Order is $O(T_1 \lg \hat{k})$.

Now we consider how queries impact the span of race detection. Given a single node u along the span of G_N that writes to memory location l. Since F-Order needs to perform reachability queries between u and every single reader in `reader-list(l)`, assuming are at most r readers in `reader-list(l)`, the total number of queries performed when executing u is at most r. Since all the queries can be computed independently of each other, such the overall query overhead has a span of $O(\lg r \lg \hat{k})$. In the worst case, every node u along the span of G_N incurs such query overhead. Then, combining Lemma 6.4, the bound follows. □

7 Implementation and Empirical Analysis

This section briefly describe our prototype implementation of F-Order and empirically evaluates its performance. We evaluate F-Order's performance across six different benchmarks and compare it to FutureRD,[4] a state-of-the-art sequential race detector for futures [51]. FutureRD provides the best sequential running time for race detecting the structured use of futures which imposes certain programming restrictions on where the future get can occur. For race detecting general futures, the algorithm by Utterback et al. [51] has an additional $\alpha(m, n)$ overhead compared to the algorithm by Agrawal et al. [1], but the algorithm by Agrawal et al. [1] has never been implemented.

[4]The codebase of FutureRD can be obtained at https://github.com/wustl-pctg/futurerd.

bench	*N*	*B*	*reads*	*writes*	*futures*	*strands*	*avg*
sw-sf	2048	64	8.59×10^9	4.20×10^6	1024	2054	1.0
hw-sf	10 (images)	-	1.73×10^{10}	1.64×10^8	3672	9914	14.05
sort	10^7	8192	2.75×10^8	2.22×10^8	14463	60030	2.95
mm	2048	64	1.72×10^{10}	1.43×10^8	18724	79577	20.94
smm	2048	64	9.40×10^8	2.50×10^5	16387	70822	8.24
ferret	simlarge	-	5.40×10^9	6.23×10^8	256	1280	13.09
sw-gf	2048	64	8.59×10^9	4.20×10^6	1024	5124	1.0
hw-gf	10 (images)	-	1.73×10^{10}	1.64×10^8	4590	11750	13.1

Figure 2. The characteristics of the benchmarks. The `sw` and `hw` benchmarks have two implementations: structured(`sf`) and general futures(`gf`).

Empirical results indicate that, even though our parallel race detector incurs higher overhead on one-core execution, the overhead is never more than 3× compared to FutureRD (in fact much less for all benchmarks except for `hw`). Thus, the fact that we can race detect while executing the program in parallel quickly pays off in absolute execution times.

Implementation. We have implemented F-Order by extending the Cilk-F runtime system [43], a work-stealing runtime system that supports the use of futures. In the implementation of F-Order, we employed WSP-Order [50] for maintaining and querying the reachability between two nodes that are in the same SP DAG (i.e., $\prec_d$ and $\|^d_{left}$). We implemented the augmentation necessary in the Cilk-F runtime as proposed by Utterback et al. [50]. As discussed in Section 6, such an augmentation is necessary in order to provide the desired performance bound. The construction functions of the FOM are called via instrumentation inserted into the parallel control constructs of Cilk-F. The instrumentation on memory accesses are inserted via ThreadSanitizer [42] pass implemented in LLVM.

Benchmarks. Six benchmarks are used: matrix multiplication (`mm`), the Strassen matrix multiplication algorithm (`smm`), parallel merge sort (`sort`), the Heart Wall application (`hw`) from the Rodinia suite [11], Smith-Waterman for sequence alignment (`sw`), and a content-based image similarity search modified from the PARSEC benchmark suite (`ferret`) [6]. We have implemented all the benchmarks with structured use of futures. The `sw` and `hw` benchmarks, in addition, have a second implementation with a general use of futures that imposes no restrictions.

FutureRD distinguishes between structured and general use of futures, and provides better running time for structureed use of futures. Our race detector does not distinguish between structured versus general use of futures and provides the same bound for both uses.

The characteristics of these benchmarks are shown in Figure 2, including the input sizes and serial base case sizes used. We also measured the average group sizes; the measurement indicates that average group sizes (related to $\hat{k}$) are indeed small across benchmarks.

Experimental Setup. All experiments were run on a machine with two 20-core Intel Xeon Gold 6148 processors,

bench	*configuration*	T_1	T_{20}	*FutureRD*
sw (structured)	baseline	21.88	2.26 [9.67×]	21.57
	reachability	22.49 (1.02×)	2.29 [9.79×]	21.52 (0.99×)
	full	697.06 (31.85×)	96.96 [7.18×]	562.55 (26.08×)
hw (structured)	baseline	15.41	0.98 [15.60×]	15.5
	reachability	17.95 (1.16×)	1.03 [17.32×]	15.5 (1.0×)
	full	943.33 (61.18×)	72.12 [13.07×]	381.85 (24.63×)
sort (structured)	baseline	1.33	0.07 [17.17×]	1.34
	reachability	5.05 (3.77×)	0.38 [13.04×]	1.34 (1.0×)
	full	30.62 (22.87×)	2.2 [13.92×]	19.48 (14.48×)
mm (structured)	baseline	8.48	0.43 [19.30×]	8.47
	reachability	12.97 (1.52×)	0.68 [19.05×]	8.47 (1.0×)
	full	484.48 (57.13×)	24.44 [19.81×]	320.46 (37.86×)
smm (structured)	baseline	2.42	0.14 [16.60×]	-
	reachability	2.84 (1.17×)	0.16 [17.30×]	-
	full	51.32 (21.14×)	2.69 [19.01×]	-
ferret (structured)	baseline	7.45	0.65 [11.41×]	7.33
	reachability	7.64 (1.02×)	0.66 [11.54×]	7.28 (0.99×)
	full	337.23 (45.26×)	32.76 [10.29×]	298.18 (40.68×)
sw (general)	baseline	21.79	2.28 [9.54×]	21.64
	reachability	24.93 (1.14×)	2.28 [10.91×]	21.58 (0.99×)
	full	704.72 (32.32×)	100.1 [7.04×]	610.75 (28.22×)
heartwall (general)	baseline	15.42	0.99 [15.48×]	15.5
	reachability	18.56 (1.2×)	1.08 [17.09×]	15.5 (1.0×)
	full	934.47 (60.6×)	68.13 [13.71×]	488.13 (31.46×)

Figure 3. Performance of the benchmarks with F-Order and FutureRD for race detection. Execution time on P processors, T_P, is given in seconds. Numbers in the parentheses show the overhead compared to the baseline. Numbers in the brackets show the scalability relative to T_1 of the same configuration. Measurements of `smm` running with FutureRD is not available because it segfaulted.

clocked at 2.40 GHz, with hyperthreading disabled. Each core has separate private 32 KB L1 data and 32 KB L1 instruction caches, and a 1 MB private L2 cache. Each socket has a 27.5 MB shared L3 cache. The machine has 768 GB of main memory. We limit execution to the first 20 cores, located on the first socket of the machine, in order to avoid NUMA overhead. All software is compiled with LLVM/Clang 3.4.1 with `-O2` optimizations running on Linux kernel version 4.15. Each data point is the average of 3 runs.

For each benchmark, we ran three different configurations: ***baseline***, where the benchmark is compiled without any race detection enabled; ***reachability***, where the benchmark is compiled with only the reachability component but not the access history; and ***full***, where the benchmark is compiled with full race detection.

Practical Performance of F-Order

Figure 3 shows our measurements for the benchmarks. With the exception of the `sort` benchmark, we see that the the reachability versions of F-Order incur very little overhead when compared to the baseline versions. The overhead is more pronounced in `sort` because a majority of the futures created do little more than generate more futures,[5] and even in the serial base case the work is only $\theta(B \lg B)$, where B is the problem size of the serial base case. Moreover, we expect the reachability overhead of FutureRD is less than that of F-Order on benchmarks using structured futures because its reachability algorithm designed for structured futures is more efficient than its algorithm designed for general futures. F-Order, however, cannot take advantage of the restrictions imposed by structured use of futures.

Full race detection versions incur a large increase in overhead in both F-Order and FutureRD, which comes from the combination of the memory instrumentation and the sheer quantity of reachability queries. The additional overhead of full race detection in F-Order compared to FutureRD is the price one pays to enable parallel race detection. In F-Order, each query incurs $O(\lg \hat{k})$ instead of constant time (which is the case for FutureRD). This overhead is the most evident in `hw`; this is because the structure of the parallelism and the memory access pattern cause significantly more non-SP queries than in any of our other benchmarks. Even in the case of the high overhead `hw`, however, the full race detection version of the benchmarks maintain scalability comparable to that of the baselines. The higher overhead of non-SP queries in F-Order can be offset by the scalability that F-Order gains. As shown in Figure 3, the absolute running times of F-Order on 20 cores are significantly faster than the running time of FutureRD. In our evaluation, the running times of all benchmarks with F-Order on 4 cores or more can beat the running time of FutureRD.

8 Conclusion

In this paper, we propose a parallel race detection algorithm for task parallel programs that employ futures. In the literature, researchers have distinguished use of futures to be structured versus general in the context of studying performance bounds [18, 43]. Moreover, Utterback et al. [51] showed that, when race detecting sequentially at least, one can exploit the structured use of futures to obtain lower race detection overhead. Unfortunately, their algorithms cannot be parallelized easily. Our proposed algorithm does not currently exploit the structured use of futures to gain a performance advantage. As an interesting research direction, we plan to investigate whether it's possible to obtain a more efficient parallel race detection algorithm when one restricts to only structured use of futures.

Acknowledgements

This research was supported in part by National Science Foundation under grants CCF-1527692, CCF-1733873, and CCF-1910568. We thank the reviewers and our shepherd for their excellent comments.

[5] This is also true for `mm` and `smm`; their serial base cases, however, perform much more work, $\theta(B^3)$.

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. 1998. Thread Scheduling for Multiprogrammed Multiprocessors. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*. 119–129.

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

[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] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, Pittsburgh, PA, USA, 332–348. https://doi.org/10.1145/2814270.2814303

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

[7] Robert D. Blumofe and Charles E. Leiserson. 1994. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. 356–368.

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

[9] 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.

[10] 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.

[11] 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.

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

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

[18] 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

[19] Chun-Hung Hsiao, Satish Narayanasamy, Essam Muhammad Idris Khan, Cristiano L. Pereira, and Gilles A. Pokam. 2017. AsyncClock: Scalable Inference of Asynchronous Event Causality. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, Xi'an, China, 193–205. https://doi.org/10.1145/3037697.3037712

[20] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, Edinburgh, United Kingdom, 326–336. https://doi.org/10.1145/2594291.2594330

[21] 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

[22] ISOIEC14882 2012. ISO/IEC 14882:2011(E) Information technology — Programming Languages — C++. Third Edition, 2012-02-14.

[23] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, Barcelona, Spain, 157–170. https://doi.org/10.1145/3062341.3062374

[24] Alex Kogan and Maurice Herlihy. 2014. The Future(s) of Shared Data Structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14)*. ACM, Paris, France, 30–39. http://doi.acm.org/10.1145/2611462.2611496

[25] I-Ting Angelina Lee and Tao B. Schardl. 2015. Efficiently Detecting Races in Cilk Programs That Use Reducer Hyperobjects. In *SPAA '15: Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, Portland, Oregon, USA, 111–122. http://doi.acm.org/10.1145/2755573.2755599

[26] Peng Liu, Omer Tripp, and Xiangyu Zhang. 2016. IPA: Improving Predictive Analysis with Pointer Analysis. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, Saarbrücken, Germany, 59–69. https://doi.org/10.1145/2931037.2931046

[27] 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.

[28] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, Edinburgh, United Kingdom, 316–325. https://doi.org/10.1145/2594291.2594311

[29] 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.

[30] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: A Lock-Free Queue with Batching. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. ACM, Vienna, Austria, 99–109.

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

[32] 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*.

[33] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming (PPoPP '03). ACM, New York, NY, USA, 167–178.

[34] Oracle. 2018. Java Platform Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/ The Future library is located in the java.util.concurrent.

[35] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, Beijing, China, 251–262. https://doi.org/10.1145/2254064.2254095

[36] Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. (2003), 179–190.

[37] 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.

[38] 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.

[39] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, Indianapolis, Indiana, USA, 151–166. https://doi.org/10.1145/2509136.2509538

[40] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *Proceedings of the Third International Conference on NASA Formal Methods (NFM'11)*. Springer-Verlag, Pasadena, CA, 313–327. http://dl.acm.org/citation.cfm?id=1986308.1986334

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

[42] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. ACM, New York, New York, 62–71.

[43] 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

[44] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[45] 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

[46] Rishi Surendran and Vivek Sarkar. 2016. Automatic Parallelization of Pure Method Calls via Conditional Future Synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 20–38. https://doi.org/10.1145/2983990.2984035

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

[48] 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.

[49] Robert Utterback. 2019. https://github.com/wustl-pctg/futurerd. Accessed in August 2019.

[50] 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.

[51] 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.

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

[53] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, Tampa Bay, FL, USA, 70–82.

[54] Caleb Voss, Tiago Cogumbreiro, and Vivek Sarkar. 2019. Transitive Joins: A Sound and Efficient Online Deadlock-avoidance Policy. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 378–390. https://doi.org/10.1145/3293883.3295724

[55] 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.

[56] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 221–234.