Dynamic Determinacy Race Detection for Task Parallelism with Futures

Rishi Surendran and Vivek Sarkar
Rice University, Houston, TX
{rishi,vsarkar}@rice.edu

Abstract. Existing dynamic determinacy race detectors for task-parallel programs are limited to programs with strict computation graphs, where a task can only wait for its descendant tasks to complete. In this paper, we present the first known determinacy race detector for non-strict computation graphs with futures. The space and time complexity of our algorithm are similar to those of the classical SP-bags algorithm, when using only structured parallel constructs such as spawn-sync and async-finish. In the presence of point-to-point synchronization using futures, the complexity of the algorithm increases by a factor determined by the number of future task creation and get operations as well as the number of non-tree edges in the computation graph. The experimental results show that the slowdown factor observed for our algorithm relative to the sequential version is in the range of $1.00 \times - 9.92 \times$, which is very much in line with slowdowns experienced for strict computation graphs in past work.

1 Introduction

Current dynamic determinacy race detection algorithms for task parallelism are limited to parallel constructs in which a task may synchronize with the parent task [15, 2], ancestor task [24, 25] or with the immediate left sibling [12]. However, current parallel programming models include parallel constructs that support more general synchronization patterns. For example, the OpenMP depends clause allows tasks to wait on previously spawned sibling tasks and the future construct in C#, C++11, Habanero Java (HJ), X10, and other languages, enables a task to wait on any previously created task to which the waiter task has a reference. Race detection algorithms based on vector clocks [1, 16] are impractical for these constructs because either the vector clocks have to be allocated with a size proportional to the maximum number of simultaneously live tasks (which can be unboundedly large) or precision has to be sacrificed by assigning one clock per processor or worker thread, thereby missing potential data races when two tasks execute on the same worker.

The approaches in [15, 2, 24, 25] focus on a structured task-parallel model, in which tasks communicate through side effects on shared variables. In contrast, our paper focuses on enabling the use of futures for functional-style parallelism, while also allowing futures to co-exist with imperative async-finish parallelism. The addition of point-to-point synchronization for futures makes the race detection more challenging than for async-finish task parallelism since the computation graphs that can be generated using futures are more general than those
that can be generated by fork-join parallel constructs such as async-finish constructs in X10 [14] and Habanero-Java [7], spawn-sync constructs in Cilk [4], and task-taskwait constructs in OpenMP [22].

Existing algorithms for detecting determinacy races for dynamic task parallelism, do not support race detection for futures. For instance, Cilk data race detectors [15, 2] handle only spawn-sync constructs where the computation graph is a Series-Parallel (SP) dag. Although the computation graphs for async-finish parallelism [24, 25] are more general than SP-dags, whether two instructions may logically execute in parallel can still be determined efficiently by a lookup of the lowest common ancestor of the instructions in the dynamic program structure tree [24, 25]. Dimitrov et.al [12] extended the SP-bags algorithm to work for 2D-lattices by formulating the race detection problem as the suprema (least upper bound) computation on the computation graph. The computation graphs in the presence of futures may not have any of the structures discussed above, and therefore, the past approaches are not directly applicable to parallel programs with futures. However, parallel programs written with futures enjoy the property that data race freedom implies determinacy, i.e., if a parallel program is written using only async, finish, and future constructs, and is known to not exhibit a data race, then it must be determinate.

The main contributions of this paper are as follows:

1. The first known sound and precise on-the-fly algorithm for detecting races in programs containing async, finish and future parallel constructs. Instead of using brute force approaches such as building the transitive closure of the happens-before relation, our algorithm relies on a novel data structure called the dynamic task reachability graph to efficiently detect races in the input program. We show that the algorithm can detect determinacy races by effectively analyzing all possible executions for a given input. Relative to the SP-bags and related algorithm, the complexity of our algorithm only increases by a factor determined by the number of future task creation and get operations as well as the number of non-tree edges in the computation graph.

2. An implementation and evaluation of the algorithm on programs with structured parallelism and point-to-point synchronization. We implemented the algorithm in the Habanero Java compiler and runtime system, and evaluated it on a suite of benchmarks containing async, finish and future constructs. The experiments show that the algorithm performs similarly to SP-bags in the presence of structured synchronization and degrades gracefully in the presence of point-to-point synchronization.

The remainder of the paper is organized as follows. Section 2 discusses our programming model, and Section 3 defines a determinacy race for our programming model. Section 4 presents the algorithm for determinacy race detection for parallel programs with futures, and Section 5 discusses the implementation and experimental results for our race detection algorithm. Section 6 discusses related work, and Section 7 contains our conclusions. Appendix A discusses the conditions for determinism and deadlock-freedom for our programming model.

1 A summary abstract of this approach was presented at SPAA 2016 [26].
Appendix B presents the proofs for the complexity and correctness of our algorithm.

2 Programming Model

Our work addresses parallel programming models that can support combinations of functional-style futures and imperative-style tasks; examples include the X10 [14], Habanero Java [7], Chapel [8], and C++11 languages. We will use X10 and Habanero Java’s finish and async notation for task parallelism in this paper, though our algorithms are applicable to other task-parallel constructs as well. In this notation, the statement “async { S }” causes the parent task to create a new child task to execute S asynchronously (i.e., before, after, or in parallel) with the remainder of the parent task. The statement “finish { S }” causes the parent task to execute S and then wait for the completion of all asynchronous tasks created within S. Each dynamic instance $T_A$ of an async task has a unique Immediately Enclosing Finish (IEF) instance $F$ of a finish statement during program execution, where $F$ is the innermost finish containing $T_A$. There is an implicit finish scope surrounding the body of main() so program execution will end only after all async tasks have completed.

A future [17] (or promise [19]) refers to an object that acts as a proxy for a result that may initially be unknown, because the computation of its value may still be in progress as a parallel task. In the notation used in this paper, the statement, “future<$T$> $f = \text{async<$T$>} \text{Expr;}$” creates a new child task to evaluate Expr asynchronously, where $T$ is the type of the expression Expr. In this case, $f$ contains a handle to the return value (future object) for the newly created task and the operation $f.get()$ can be performed to obtain the result of the future task. If the future task has not completed as yet, the task performing the $f.get()$ operation blocks until the result of Expr becomes available. Futures are traditionally used for enabling functional-style parallelism and are guaranteed not to exhibit data races in their return values. However, imperative programming languages allow future tasks to also contain side effects in the task bodies. These side effects on shared memory locations may cause determinacy races if the program has insufficient synchronization. Another use case for futures is a future task with void return values, where the get() operation is used purely for point-to-point synchronization and the tasks only communicate values through side effects, just as with async and finish constructs.

**Comparison with spawn-sync and async-finish** In both spawn-sync and async-finish programming models, a join operation can be performed only once on a task (by the parent task in spawn-sync and by the ancestor task containing the immediately enclosing finish in async-finish). The class of computations generated by spawn-sync constructs is said to be fully strict, and the class of computations generated by async-finish constructs is called terminally strict.

The introduction of future as a parallel construct increases the possible synchronization patterns. Task $T_2$ can wait for a previously created task $T_1$ if $T_2$ has a reference to $T_1$ by performing the get() operation. Moreover, this join operation on task $T_1$ can be performed by multiple tasks. As an example, consider
the program in Figure 1, where the main program creates three future tasks $T_A$, $T_B$, and $T_C$. There are three join operations on task $T_A$ performed by sibling tasks $T_B$, $T_C$, and the parent task. Here Stmt3, Stmt6, and Stmt8 may execute in parallel with task $T_A$, while Stmt4, Stmt7, and Stmt9 can execute only after the completion of task $T_A$. Synchronization using `get()` can lead to transitive dependences among tasks. For example, although the main task in Figure 1 did not perform an explicit join on task $T_B$, there is a transitive join dependence from $T_B$ to the main task, because task $T_C$ performed a get operation on task $T_B$ due to which Stmt10 can execute only after tasks $T_A$, $T_B$, and $T_C$ complete their execution.

```cpp
1 // Main task
2 Stmt1;
3 future<T> A = async<T> { ... }; // Task $T_A$
4 Stmt2;
5 future<T> B = async<T>{ Stmt3; A.get(); Stmt4; }; // Task $T_B$
6 Stmt5;
7 future<T> C = async<T>{ Stmt6; A.get(); Stmt7; B.get(); }; // Task $T_C$
8 Stmt6;
9 A.get();
10 Stmt7;
11 C.get();
12 Stmt10;
```

Fig. 1: Example Program with HJ Futures. $A, B$ and $C$ hold references to future tasks created by the main program

### 3 Data Races and Determinacy

In this section, we formalize the definition of data races in programs containing async, finish, and future constructs as a preamble to defining determinacy races. Our definition extends the notion of a computation graph [5] for a dynamic execution of a parallel program, in which each node corresponds to a step which is defined as follows:

**Definition 1.** A step is a maximal sequence of statement instances such that no statement instance in the sequence includes the start or end of an async, finish or a get operation.

The edges in a computation graph represent different forms of happens-before relationships. For the constructs covered in this paper (async, finish, future), there are three different types of edges:

1. **Continue Edges** capture the sequencing of steps within a task. All steps in a task are connected by continue edges.
2. **Spawn Edges** represent the parent-child relationship among tasks. When task A creates task B, a spawn edge is inserted from the step that ends with the async in task A to the step that starts task B.
3. **Join Edges** represent the synchronization among tasks. When task A performs a get on future B, a join edge is inserted from the last step of B to the step in task A that immediately follows the `get()` operation. Join edges are also inserted from the last step of every task to the step in the ancestor task immediately following the Immediately Enclosing Finish (IEF). A join
Fig. 2: Example program with futures and its computation graph. S1-S12 are steps in the program. The circles represent the steps in the program. The rectangles represent tasks. $T_M$ is the main task and $T_A$, $T_B$, $T_C$ and $T_D$ are future tasks created during the execution of the program.

edge from task B to task A is referred to as tree join if A is an ancestor of B; otherwise, it is referred to as a non-tree join.

All three kinds of edges have been defined in past work on computation graphs for the Cilk [4] and Habanero-Java languages, except for non-tree join edges.

Definition 2. A step $u$ is said to precede step $v$, denoted as $u \prec v$, if there exists a path from $u$ to $v$ in the computation graph.

We use the notation $Task(u) = T$ to indicate that step node $u$ belongs to task $T$, and $u \not\prec v$ to denote the fact that there is no path from step $u$ to step $v$ in the computation graph. Two distinct steps, $u$ and $v$ operate logically in parallel, denoted $u \parallel v$, if $u \not\prec v$ and $v \not\prec u$.

Definition 3. A data race may occur between steps $u$ and $v$, iff both $u$ and $v$ access a common memory location, at least one of which is a write and $u \parallel v$.

As an example, consider the program in Figure 2 which creates four future tasks: $T_A$, $T_B$, $T_C$, and $T_D$. S1-S12 represent the steps in the program. Here $S2 \not\prec S10$ because there is no directed path from $S2$ to $S10$ in the computation graph, and $S2 \prec S12$ since there is a directed path from $S2$ to $S12$. The join edge from $S3$ to $S5$ is a tree join since $T_A$ is an ancestor of $T_B$. The edge from $S5$ to $S8$ is a non-tree join since $T_C$ is not an ancestor of $T_A$.

We say that a parallel program is functionally deterministic if it always computes the same answer when given the same inputs. Further, we refer to a program as structurally deterministic if it always computes the same computation graph, when given the same inputs. Finally, following past work [18, 10], we say that a program is determinate if it is both functionally and structurally deterministic. If a parallel program is written using only async, finish, and future constructs, and is guaranteed to never exhibit a data race, then it must be determinate, i.e., both functionally and structurally deterministic. Note that all data-race-free programs written using async, finish and future constructs are guaranteed to be determinate, but it does not imply that all racy programs are non-determinate. For instance, a program with parallel writes of the same value to a common memory location is racy, yet determinate.
4 Determinacy Race Detection Algorithm

In this section, we present our algorithm for detecting determinacy races in programs with async, finish and future as parallel constructs. A dynamic determinacy race detector needs to provide mechanisms that answers two questions: for any pair of memory accesses, at least one of which is a write, 1) can the two accesses logically execute in parallel?, and 2) do they access the same memory location? To answer the first question, we introduce a program representation referred to as dynamic task reachability graph which is presented in Section 4.1. Similar to most race detectors, we use a shadow memory mechanism to answer the second question which is presented in Section 4.2. Section 4.3 presents our determinacy race detection algorithm.

4.1 Dynamic Task Reachability Graph

Since storing the entire computation graph of the program execution is memory inefficient, we use a more compact representation while still retaining sufficient information to precisely answer all reachability queries during race detection. Our program representation, referred to as a dynamic task reachability graph, represents reachability information at the task-level instead of the step-level. This significantly reduces the storage requirements; however, our algorithm can still precisely detect determinacy races in parallel programs with futures. Note that the representation assumes that the input program is executed serially in depth-first order. Our representation uses the following three ideas for encoding reachability information between steps in the computation graph of the input program:

Disjoint set representation of tree joins The reachability information between tasks which are connected by tree join edges is represented using a disjoint set data structure. Two tasks A and B are in the same set if and only if B is a descendant of A and there is a path in the computation graph from B to A which includes only tree-join edges and continue edges. Similar to the SP-bags algorithm, our algorithm uses the fast disjoint-set data structure [9, Chapter 22], which maintains a dynamic collection of disjoint sets \( \Sigma \) and provides three operations:

1. \text{MakeSet}(x) which creates a new set that contains \( x \) and adds it to \( \Sigma \)
2. \text{Union}(X,Y) which performs a set union of \( X \) and \( Y \), adds the resulting set to \( \Sigma \) and destroys set \( X \) and \( Y \)
3. \text{FindSet}(x) which returns the set \( X \in \Sigma \) such that \( x \in X \).

Any \( m \) of these three operations on \( n \) sets takes a total of \( O(m \alpha(m,n)) \) time [27]. Here \( \alpha \) is Tarjan’s functional inverse of Ackermann’s function which, for all practical purposes is bounded above by 4.

Interval encoding of spawn tree In order to efficiently store and answer reachability information from a task to its descendants, we use a labeling scheme [11], in which each task is assigned a label according to preorder and postorder numbering schemes. The values are assigned according to the order in which the tasks are visited during a depth-first-traversal of the spawn tree, where the nodes in the spawn tree corresponds tasks and edges represent the
parent-child spawn relationship. Using this scheme, the ancestor-descendant relationship queries between task pairs can be answered by checking if the interval of one task subsumes the interval of the other task. For example, if \([x, \text{pre}, x, \text{post}]\) is the interval associated with task \(x\) and \([y, \text{pre}, y, \text{post}]\) is the interval associated with task \(y\), then \(x\) is an ancestor of \(y\) if and only if \(x, \text{pre} \leq y, \text{pre}\) and \(y, \text{post} \leq x, \text{post}\). When a task \(B\) joins with the ancestor task \(A\), the disjoint sets of \(A\) and \(B\) are merged together and the new set will have the label originally associated with \(A\). Although, a label is assigned to every task when it is spawned, the labels are associated with each disjoint set in general. Compared to past work [11] which used labeling schemes on static trees, the tree is dynamic in our approach since race detection is performed on-the-fly. This requires a more general labeling scheme, where a temporary label is assigned when a task is spawned and the label is updated when the task returns to its parent.

**Immediate predecessors + significant ancestor representation of non-tree joins** The non-tree joins in the computation graph are represented in the dynamic task reachability graph as follows:

- **Immediate predecessors**: For each non-tree join from task \(A\) to task \(B\), \(B\) stores \(A\) in its set of predecessors.
- **Lowest significant ancestor**: We define the significant ancestors of task \(A\) as the set of ancestors of \(A\) which has performed at least one non-tree join operation. For each task, we store only the lowest significant ancestor.

![Dynamic Task Reachability Graph Example](image)

**Definition 1.** A dynamic task reachability graph of a computation graph \(G\) is a 5-tuple \(R = (N, D, L, P, A)\), where

- \(N\) is the set of vertices, where each vertex represents a task.
- \(D = \{D_i\}_{i=1}^n\) is a partitioning of the vertices in \(N\) into disjoint sets, \(\bigcup_{i=1}^n D_i = N\). Each partition consists of tasks which are connected by tree-join edges.
- \(L : N \to \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0}\) is a map from vertices to their labels, where each label consists of the preorder and postorder value of the vertex in the spawn tree. A label is also associated with each disjoint set \(D_i \in D\), where the label for
\( D_i \) is same as the label of \( u \), where \( u \in D_i \) and \( u \) is the node in \( D_i \) that is closest to the root of the spawn tree.

- \( P : N \to 2^N \) represents the set of non-tree edges \( P(u) = \{v_1, \ldots, v_k\} \) if and only if there are non-tree join edges from tasks \( v_1 \ldots v_k \) to \( u \).

- \( A : N \to N \) represents the lowest ancestor with at least one incoming non-tree edge.

\[ A(u) = v, \quad \text{if and only if} \quad w_1, w_2, \ldots, w_m (\text{where } r = w_1, v = w_k \text{ and } u = w_m) \text{ is the path consisting of spawn edges from the root } r \text{ of } G \text{ to } u, \]

\[ P(w_j) = \emptyset, \quad \forall j \text{ such that } k + 1 \leq j \leq m - 1 \text{ and } P(v) \neq \emptyset. \]

\( v \) is referred to as the lowest significant ancestor (LSA) of \( u \).

### Table 1

(a) is the dynamic task reachability graph for the computation graph in Figure 3 after execution of step 11. Task \( T_3 \) performed join operations on \( T_2 \) and \( T_1 \). Therefore \( P(T_3) = \{T_1, T_2\} \). The least significant ancestor of \( T_4, T_5 \) and \( T_6 \) is \( T_3 \) because \( T_3 \) is their lowest ancestor which performed a non-tree join. (b) is the dynamic task reachability graph for the computation graph in Figure 3 after execution of step 17. \( T_0, T_3, T_4, T_5 \) and \( T_6 \) are all in the same disjoint set because they are connected by tree join edges.

### 4.2 Shadow Memory

Our algorithm maintains a shadow memory \( M_s \) for every shared memory location \( M \). \( M_s \) contains the following fields

- \( w \), a reference to a task that wrote to \( M \). \( M_s.w \) is initialized to \( \text{null} \) and is updated at every write to \( M \). It refers to the task that last wrote to \( M \).

- \( r \), a set of references to tasks that read \( M \). \( M_s.r \) is initialized to \( \emptyset \) and is updated at reads of \( M \). It contains references to all future tasks that read \( M \) in parallel, since the last write to \( M \). It also contains a reference to one non-future (async) task which read \( M \) since the last write to \( M \).

### 4.3 Algorithm

The overall determinacy race detection algorithm is given in Algorithms 1-10. As the input program executes in serial, depth-first order the race detection algorithm performs additional operations whenever one of the following actions...
### Algorithm 1: Initialization

<table>
<thead>
<tr>
<th>Input:</th>
<th>Main task M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>dfid ← 0</td>
</tr>
<tr>
<td>2:</td>
<td>tmpid ← MAXINT</td>
</tr>
<tr>
<td>3:</td>
<td>SM ← Make-Set(M)</td>
</tr>
<tr>
<td>4:</td>
<td>SM.pre ← dfid</td>
</tr>
<tr>
<td>5:</td>
<td>dfid ← dfid + 1</td>
</tr>
<tr>
<td>6:</td>
<td>SM.post ← tmpid</td>
</tr>
<tr>
<td>7:</td>
<td>tmpid ← tmpid − 1</td>
</tr>
<tr>
<td>8:</td>
<td>SM.parent ← null</td>
</tr>
<tr>
<td>9:</td>
<td>SM.lsa ← null</td>
</tr>
</tbody>
</table>

### Algorithm 2: Task creation

<table>
<thead>
<tr>
<th>Input:</th>
<th>Parent task P, Child task C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>SC ← Make-Set(C)</td>
</tr>
<tr>
<td>2:</td>
<td>SC.pre ← dfid</td>
</tr>
<tr>
<td>3:</td>
<td>dfid ← dfid + 1</td>
</tr>
<tr>
<td>4:</td>
<td>SC.post ← tmpid</td>
</tr>
<tr>
<td>5:</td>
<td>tmpid ← tmpid − 1</td>
</tr>
<tr>
<td>6:</td>
<td>SC.parent ← Sp</td>
</tr>
<tr>
<td>7:</td>
<td>if Sp.nt = {} then</td>
</tr>
<tr>
<td>8:</td>
<td>SC.lsa ← Sp.lsa</td>
</tr>
<tr>
<td>9:</td>
<td>else</td>
</tr>
<tr>
<td>10:</td>
<td>SC.lsa ← Sp</td>
</tr>
<tr>
<td>11:</td>
<td>end if</td>
</tr>
</tbody>
</table>

occurs: task creation, task return, get() operation, shared memory read and shared memory write. The race detector stores the following information associated with every disjoint set of tasks.

- pre and post together form the interval label assigned to the disjoint set.
- nt is the set of incoming non-tree edges.
- parent refers to the parent task.
- lsa represents the least significant ancestor.

Next, we describe the actions performed by our race detector:

**Initialization:** Algorithm 1 shows the initialization performed by our race detector when the main task \( M \) is created. The set \( S_M \) is initialized to contain task \( M \). It assigns \([0, \text{MAXINT}]\) as the interval label for the main task. Since the postorder value of a node is known only after the full tree has unfolded, we assign a temporary postorder value MAXINT (the largest integer value). The parent and lsa fields are initialized to null.

**Task Creation:** Algorithm 2 shows the actions performed by our race detector during task creation. Whenever a task \( P \) spawns a new task \( C \), \( C \) is assigned the preorder value and a temporary postorder value. Our algorithm assigns temporary postorder values starting at the largest integer value (MAXINT) in decreasing order. This assignment scheme maintains the interval label property, where the label of an ancestor subsumes the labels of descendants. The set \( S_C \) is initialized to contain task \( C \). The least significant ancestor for task \( C \) is initialized at task creation time based on whether task \( P \) has performed any non-tree joins.

**Task Termination:** When task \( C \) terminates, the postorder value of \( C \) is updated with the final value. This is shown in Algorithm 3.

**Get Operation:** Algorithm 4 shows the actions performed by the race detector at a get() operation. When task \( A \) performs a get() operation on task \( B \), there are two possible cases: 1) \( A \) is an ancestor of \( B \) and there are join edges from all tasks which are descendants of \( A \) and ancestors of \( B \) to \( A \). In this case, the algorithm performs a union of the disjoint sets \( S_A \) and \( S_B \) by invoking the Merge function given in Algorithm 7, and 2) there is a non-tree join edge from \( B \) to \( A \). In this case, \( B \) is added to the sequence of non-tree predecessors of \( A \).

**Finish:** Algorithm 5 and Algorithm 6 show the actions performed by the race detector at the start and end of a finish. At the end of a finish \( F \), the disjoint
Algorithm 3: Task termination

**Input:** Terminating task \( C \)
1: \( S_C.post \leftarrow dfid \)
2: \( dfid \leftarrow dfid + 1 \)
3: \( tmpid \leftarrow tmpid + 1 \)

Algorithm 4: Get operation

**Input:** Tasks \( A, B \) where \( A \) is performing \( B \)
1: if \( \text{Find-Set}(A) = \text{Find-Set}(B.parent) \) then
2: \( \text{Merge}(S_A, S_B) \)
3: else
4: \( S_A.nt \leftarrow S_A.nt \cup \{ B \} \)
5: end if

Algorithm 5: Start finish

**Input:** Start of finish scope \( F \) within task \( A \)
1: \( F.parent \leftarrow A \)

Algorithm 6: End finish

**Input:** Finish \( F \)
1: \( A \leftarrow F.parent \)
2: for \( B \in F.joins \) do
3: \( \text{MERGE}(S_A, S_B) \)
4: end for

Algorithm 7: Merge tasks

**Input:** Disjoint sets \( S_A, S_B \)
1: procedure \( \text{Merge}(S_A, S_B) \)
2: \( nt \leftarrow S_A.nt \cup S_B.nt \)
3: \( lsa \leftarrow S_A.lsa \)
4: \( S_A \leftarrow S_B \cup \text{UNION}(S_A, S_B) \)
5: \( S_A.nt \leftarrow nt \)
6: \( S_A.lsa \leftarrow lsa \)
7: end procedure

Algorithm 8: Get operation

Shared Memory Access: Determinacy races are detected when a read or write to a shared memory location occurs. When a write to a memory location \( M \) is performed by step \( u \), the algorithm checks if the previous writer or the previous readers in the shadow memory space may execute in parallel with the currently executing step and reports a race. It updates the writer shadow space of \( M \) with the current task and removes any reader \( r \) if \( r \prec u \). This is shown in Algorithm 8. When a read to a memory location \( M \) is performed by step \( u \), the algorithm checks if the previous writer in the shadow memory space may execute in parallel with the currently executing step and reports a race. It adds the current task to the set of readers of \( M \) and removes any task \( r \) if \( r \prec u \). Our algorithm differentiates between future tasks and async tasks: async tasks can be waited upon by only ancestor tasks using the finish construct and future tasks can be waited upon using the get() operation. Given a task \( A \) as argument, \( \text{IsFuture} \) returns true, if \( A \) is a future task. The readers shadow memory contains a maximum of one async task, but may contain multiple future tasks. During the read of a shared memory location by step \( s \) of an async task \( A \), the algorithm replaces the previous async reader \( X \) by \( A \), if \( X \) precedes \( s \). This is shown in Algorithm 9.

Given tasks \( A \) and \( B \), \text{PRECEDE} routine shown in Algorithm 10 checks if task \( A \) must precede \( B \) by invoking routine \text{VISIT} which is also given in Algorithm 10. Lines 6–11 of \text{VISIT} routine returns true if the interval corresponding to the disjoint set of \( B \) is contained in the interval corresponding to the disjoint set of \( A \). Lines 12–14 returns false, if the preorder value of \( A \) is greater than the preorder value of \( B \), since the source of a non-tree join edge must have a lower preorder value than the sink of the non-tree edge. Lines 15–20 checks if \( B \) is reachable
**Input:** Memory location $M$, Task $A$ that writes to $M$

1: for $X \in M.s.r$ do
2: if not $\text{Precede}(X, A)$ then
3: a determinacy race exists
4: else
5: $M.s.r \leftarrow M.s.r - \{X\}$
6: end if
7: end for
8: if not $\text{Precede}(M.s.w, A)$ then
9: a determinacy race exists
10: end if
11: $M.s.w \leftarrow A$

**Algorithm 8:** Write check

**Input:** Memory location $M$, Task $A$ that reads $M$

1: update $= false$
2: for $X \in M.s.r$ do
3: if $\text{Precede}(X, A)$ then
4: $M.s.r \leftarrow M.s.r - \{X\}$
5: update $\leftarrow true$
6: else if $\text{IsFuture}(X)$ or $\text{IsFuture}(A)$ then
7: update $\leftarrow true$
8: end if
9: end if
10: end for
11: if not $\text{Precede}(M.s.w, A)$ then
12: a determinacy race exists
13: end if
14: if update then
15: $M.s.r \leftarrow M.s.r \cup \{A\}$
16: end if

**Algorithm 9:** Read check

from $A$ along the immediate non-tree predecessors of $B$. Lines 21–29 traverses paths which include the non-tree predecessors of the significant ancestors of $B$ starting with the least significant ancestor of $B$. The routine returns true when a path from $A$ to $B$ is found or returns false when all the non-tree edges whose source has a preorder value greater than the preorder value of $A$ are visited.

The following two theorems discuss the complexity and correctness of our race detection. The proofs of these theorems are given in Appendix B.

**Theorem 1.** Consider a program with async, finish and future constructs that executes in time $T$ on one processor, creates $a$ async tasks, $f$ future tasks, performs $n$ non-tree join edges and references $v$ shared memory locations. Algorithms 1–10 can be implemented to check this program for determinacy races in $O(T(a + f + n + v * (f + 1))$ time using $O(a + f + n + v * (f + 1))$ space.
Here $\alpha$ is Tarjan’s functional inverse of Ackermann’s function which, for all practical purposes is bounded above by 4.

**Theorem 2.** Algorithms 1–10 detect a determinacy race in the input program if and only if a determinacy race exists.

## 5 Experimental Results

In this section, we present experimental results for our determinacy race detection algorithm. The race detector was implemented as a new Java library for detecting determinacy races in HJ programs containing async, finish and future constructs. The benchmarks written in HJ were instrumented for race detection during a bytecode-level transformation pass implemented on HJ’s Parallel Intermediate Representation (PIR) [21]. The PIR extends Soot’s Jimple IR [28] with parallel constructs such as async, finish, and future. The instrumentation pass adds the necessary calls to our race detection library at async, finish and future boundaries, future get operations, and also on reads and writes to shared memory locations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Tasks</th>
<th>#NTJoins</th>
<th>#SharedMem</th>
<th>Seq (milliseconds)</th>
<th>Racedet (milliseconds)</th>
<th>Slowdown (Racedet/Seq)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series-af</td>
<td>999,999</td>
<td>0</td>
<td>4,000,059</td>
<td>483,224</td>
<td>484,746</td>
<td>1.00</td>
</tr>
<tr>
<td>Series-future</td>
<td>999,999</td>
<td>0</td>
<td>6,000,059</td>
<td>487,134</td>
<td>487,985</td>
<td>1.00</td>
</tr>
<tr>
<td>Crypt-af</td>
<td>12,500,000</td>
<td>0</td>
<td>1,150,000,682</td>
<td>15,375</td>
<td>119,504</td>
<td>7.77</td>
</tr>
<tr>
<td>Crypt-future</td>
<td>12,500,000</td>
<td>0</td>
<td>1,175,000,682</td>
<td>15,517</td>
<td>128,234</td>
<td>8.26</td>
</tr>
<tr>
<td>Jacobi</td>
<td>8,192</td>
<td>34,944</td>
<td>641,499,805</td>
<td>3,402</td>
<td>27,388</td>
<td>8.05</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>1,608</td>
<td>4,641</td>
<td>1,652,175,806</td>
<td>3,488</td>
<td>34,558</td>
<td>9.92</td>
</tr>
<tr>
<td>Strassen</td>
<td>30,811</td>
<td>33,612</td>
<td>1,610,522,196</td>
<td>6,281</td>
<td>33,618</td>
<td>5.35</td>
</tr>
</tbody>
</table>

Table 2: Runtime overhead for determinacy race detection.

Our experiments were conducted on a 16-core Intel Ivybridge 2.6 GHz system with 48 GB memory, running Red Hat Enterprise Linux Server release 7.1, and Sun Hotspot JDK 1.7. To reduce the impact of JIT compilation, garbage collection and other JVM services, we report the mean execution time of 10 runs repeated in the same JVM instance for each data point. We evaluated the algorithm on the following benchmarks:

- **Series-af**: Fourier coefficient analysis from JGF [6] benchmark suite (Size C), parallelized using async and finish.
- **Series-future**: Fourier coefficient analysis from JGF benchmark suite (Size C), parallelized using futures.
- **Crypt-af**: IDEA encryption algorithm from JGF benchmark suite (Size C), parallelized using async and finish.
- **Crypt-future**: IDEA encryption algorithm from JGF benchmark suite (Size C), parallelized using futures.
- **Jacobi**: 2 dimensional 5-point stencil computation on a $2048 \times 2048$ matrix, where each task computes a $64 \times 64$ submatrix.

- **Strassen**: Multiplication of $1024 \times 1024$ matrices using Strassen’s algorithm. The implementation uses a recursive cutoff of $32 \times 32$.

- **Smith-Waterman**: Sequence alignment of two sequences of size 10000. The alignment matrix computation is done by $40 \times 40$ future tasks.

The first four benchmarks were derived from the original versions in the JGF suite. Jacobi and Strassen were translated by the authors from OpenMP versions of those programs in the Kastors [29] benchmark suite. The original versions of these benchmarks used the OpenMP 4.0 `depends` clause, in which tasks specify data dependence using `in`, `out` and `inout` clauses. The translated versions of these benchmarks used future as the main parallel construct, with `get()` operations used to synchronize with previously data dependent tasks. In general, this kind of task dependences cannot be represented using only async-finish constructs without loss of parallelism. The Smith-Waterman benchmarks uses futures and is based on a programming project in COMP322, an undergraduate course on parallel computing at Rice University.

The results of our evaluation is given in Table 2. The first column lists the benchmark name, and the second column shows the dynamic number of tasks (#Tasks) created for the inputs specified above. The third column shows the number of non-tree joins (#NTJoins) performed by each of the applications (the subset of future `get()` operations that are non-tree-joins). The fourth column shows the total number of shared memory accesses (#SharedMem) performed by the applications (all accesses to instance/static fields and array elements). The fifth column (#AvgReaders) shows the average number of past parallel readers per location stored in the shadow memory when a read/write access is performed on that location. (The average is computed across all accesses and all locations.) For a given access, the number of such stored readers will be either zero or one for programs containing only async and finish constructs, thereby ensuring that the average must be in the $0 \ldots 1$ range for async-finish programs. For programs with futures, the number of stored readers can be greater than one, if the location being accessed is in the read-shared state and is read by multiple tasks that can potentially execute in parallel each other. Thus, #AvgReaders can be any value that is $\geq 0$, for programs with futures.

The next column (Seq) reports the average execution time of the sequential (serial elision) version of the benchmark, and the following column (Racedet) reports the average execution time of a 1-processor execution of the parallel benchmark using the determinacy race detection algorithm introduced in this paper. Finally, the Slowdown column reports the ratio of the Racedet and Seq values.

We can make a number of observations from the data in Table 2. First, if we compute the Seq/#Tasks ratio for all the benchmarks, we can see that the Crypt-af and Crypt-future benchmarks perform $\approx 100 \times$ less work per task on average, relative to all the other benchmarks. This is the primary reason why the Crypt-af and Crypt-future benchmarks exhibit slowdowns of $7.77 \times$ and $8.26 \times$. With less work per task, the overhead per task during race detection becomes more significant than in other benchmarks; further, creating data structures for large numbers of tasks puts an extra burden on garbage collection and memory
management. However, it is important to note that the slowdowns for Series-af and Crypt-af are comparable to the slowdowns reported for the ESP-Bags algorithm [23] that only supported async and finish, thereby showing that our determinacy race detector does not incur additional overhead for async/finish constructs relative to state-of-the-art implementations.

Next, we see that the number of non-tree joins performed by Series-af and Crypt-af is zero, since they are async-finish programs for which all join (finish) operations appear as tree-join edges in the computation graph (Section 3). Since their corresponding future versions, Series-future and Crypt-future, used futures to implement async-finish synchronization, their future get() operations also appear as tree-join edges in the computation graph, thereby resulting in zero non-tree joins as well. However, the future versions of these two benchmarks have higher number of shared memory accesses than the async-finish versions, due to the additional writes and reads of future references which happened to be stored in shared (heap) locations for both benchmarks. In particular, we know that the reference to each future task must be subjected to at least one write access (when the future task is created) and one read access (when a get() operation is performed on the future), though more accesses are possible. Since Series-future creates 999,999 future tasks, we see that the difference in the #SharedMem values for Series-future and Series-af is 2,000,000 which is very close to the lower bound of $2 \times 999,999$. Likewise, for Crypt-future and Crypt-sf, the number of tasks created is 12,500,000 and the difference in the #SharedMem values is 25,000,000 which exactly matches the lower bound of $2 \times 12,500,000$. The slowdown for Crypt-future is higher than that of Crypt-af because of two reasons: 1) the additional number of memory accesses due to the future references and 2) the average number of readers stored in the shadow memory is higher, because of the presence of future tasks.

The slowdowns for Jacobi, Smith-Waterman and Strassen (8.05, 9.92, and 5.35x) are positively correlated by the values of #SharedMem, #AvgReaders, and 1/Seq, and these correlations can help explain the relative slowdowns for the three benchmarks. A larger value of #SharedMem leads to a larger slowdown due to the overhead of processing additional shared memory accesses. A larger value of #AvgReaders leads to a larger slowdown because the number of reachability queries required per shared memory access is equal to the number of readers present in the shadow memory for that location. A larger value of 1/Seq indirectly leads to a larger slowdown due to the smaller available time to amortize the overheads of race detection.

Finally, we observe that the slowdowns are not significantly impacted by the number of non-tree edges. This is because the producer and consumer tasks of a future object happen to be closely located to each other in the computation graph (for these benchmarks), usually only requiring 1-2 hops involving non-tree edges.

6 Related Work

Dynamic data race detection techniques target either structured parallelism or unstructured parallelism. Race detection for unstructured parallelism typically
uses vector clock algorithms, e.g., [1, 16]. These algorithms are impractical for task parallelism because either the vector clocks have to be allocated with a size proportional to the maximum number of simultaneously live tasks (which can be unboundedly large) or precision has to be sacrificed by assigning one clock per processor or worker thread, thereby missing potential data races when two tasks execute on the same worker.

Mellor-Crummey [20] presented Offset-Span labeling algorithm for nested fork-join constructs, which is an extension of English-Hebrew labeling scheme [13]. The idea behind their techniques is to attach a label to every thread in the program and use these labels to check if two threads can execute concurrently. The length of the labels associated with each thread is bounded by the maximum nesting depth of fork-join in the program. Our approach uses a labeling scheme which is of constant size to store reachability information between ancestor-descendant tasks. While Offset-Span labeling algorithm supports only nested fork-join constructs, our algorithm supports a more general set of computation graphs.

Feng and Leiserson [15] introduced the SP-bags algorithm for Cilk’s fully-strict parallelism, which uses only a constant factor more memory than does the program itself. Bender et al. [2] presented parallel SP-hybrid algorithm which uses English-Hebrew labels and SP-bags to detect races in Cilk programs. Despite its good theoretical bounds, the paper did not provide an implementation of the algorithm. Raman et al. [24] extended the SP-bags algorithm to support async-finish parallelism. They subsequently proposed SPD3 algorithm [25] also for async-finish parallelism, which operates in parallel. The algorithm determines series-parallel relationships between steps by a lookup of the lowest common ancestor in the dynamic program structure tree. In contrast to these approaches, our data race detection algorithm handles async, finish and futures, which can create more general computation graphs than those that can be generated by async-finish parallelism.

7 Conclusions

In this paper, we presented the first known determinacy race detector for dynamic task parallelism with futures. As with past determinacy race detectors, our algorithm guarantees that all potential determinacy races will be checked so that if a race is reported for a given input in one run of our algorithm, it will always be reported in all runs. Likewise, if no race is reported for a given input, then all executions with that input are guaranteed to be race-free. Our approach builds on a novel data structure called the dynamic task reachability graph which models task reachability information for non-strict computation graphs in an efficient manner. We presented a complexity analysis of our algorithm, and also discussed its correctness. We implemented the algorithm, and evaluated it on both strict and non-strict computations. The results indicate that the performance of our approach is similar to other efficient algorithms for spawn-sync and async-finish programs and degrades gracefully in the presence of futures. Specifically, the experimental results show that the slowdown factor observed for our algorithm relative to the sequential version is in the range of
1.00× − 9.92×, which is very much in line with slowdowns experienced for fully strict computation graphs.
References


A  Data Race Freedom, Deadlock Freedom, and Determinacy

In this section, we discuss sufficient conditions for a program containing async, finish, and future constructs to be deterministic and deadlock-free. The following program is an example of a program containing two future tasks which may deadlock in certain executions (or encounter a NullPointerException in other executions).

```java
future<T> a = null, b = null;
async { a = async<T> { b.get(); ...}; /*F1*/ }
async { b = async<T> { a.get(); ...}; /*F2*/ }
```

Deadlock can occur in the case when future tasks F1 and F2 wait for each other indefinitely via the `a.get()` and `b.get()` operations. There are multiple possible computation graphs for this program, and one of them has a cyclic dependence between tasks. In Section A.1, we describe the serial elision for a program containing async, finish, and future constructs, and in Section A.2 we show that a program without any data races as defined in Section 3 is guaranteed to be deterministic, deadlock-free and has the same semantics as the serial elision of the program. As a result, and following past conventions, we will refer to data races in programs containing async, finish and future constructs as determinacy races.

A.1  Serial Elision

Similar to spawn-sync and async-finish programs, programs with futures have a well defined serial elision version, which is the serial program after removing all the parallel constructs. To be specific, the serial elision of a program is obtained by removing all async and finish constructs and replacing “future<T> f” by “T f”, “async<T> Expr;” by “Expr;” and f.get() by f. The presence of a serial elision allows for executing the input program in depth-first order on a single processor. Our race detection algorithm makes use of this property for identifying memory accesses that can logically execute in parallel.

A.2  Data Race Freedom and Deadlock Freedom

In this section, we show that a program with async, finish, and future constructs can deadlock only if there are data races on references to futures. Let us first take a closer look at a future task creation and a get() operation on a future task. When the parent task creates a task using the statement “A = async<T> Expr;”, it performs two actions in sequence: 1) it creates the child task which is represented in the computation graph as a spawn edge and 2) updates A with the reference to the child task. Any task performing a get() operation on the child task must have the reference to the task. The following lemma presents the necessary condition for a step in a race-free program to perform a get operation.
Lemma 1. Let \((s_i, s_j)\) be a spawn edge in computation graph \(G\), where \(Task(s_i) = T_A\) and \(Task(s_j) = T_B\). Let \((s_i, s_m)\) be a continue edge in task \(T_A\). Let \(s_l\) be a step which can be executed only after a join operation on \(T_B\). Then if the input program is race free, we have \(s_m \prec s_l\).

Proof. The edge \((s_i, s_j)\) corresponds to the creation of task \(T_B\) and a join operation on task \(T_B\) has to be performed before the execution of step \(s_l\). A get() on task \(T_B\) can be performed before the execution of step \(s_l\) only if the creation of \(T_B\) is guaranteed to complete and a reference to task \(T_B\) is available at that point. In this case, step \(s_m\) writes the reference of the child task to a memory location. Therefore, we have \(s_m \prec s_l\).}

The above lemma is illustrated in Figure 4, where task \(T_a\) spawns task \(T_b\). A reference to task \(T_b\) is available at step \(s_m\) and therefore any step \(s_l\) which performs a join operation on \(T_B\) must have a path in the computation graph from \(s_m\). Next we establish a property for all edges in a race free computation graph, which in turn proves that a race free program is deadlock free.

Lemma 2. Suppose that two steps \(s_1\) and \(s_2\) execute in order in a serial, depth-first execution of a computation graph \(G\), and suppose that there exists an edge \((s_2, s_1)\) in \(G\). Then, the program has a data race.

Proof. A continue edge cannot exist from \(s_2\) to \(s_1\) because a continue edge represents the sequencing of steps within a single task. If \(s_1\) and \(s_2\) belong to the same task and \(s_1\) executes before \(s_2\) during a depth-first execution, then \(s_1 \prec s_2\).

Now, suppose there exists a spawn edge from \(s_2\) to \(s_1\). A spawn edge represents a control dependence and therefore \(s_2 \prec s_1\). This implies that \(s_2\) must execute before \(s_1\) during the serial depth-first execution. This is a contradiction of our initial assumption that \(s_1\) executes before \(s_2\) during the serial depth-first execution.

Let \(Task(s_1) = T_a\) and \(Task(s_2) = T_b\) and let us consider if it’s possible to have a join edge from \(T_b\) to \(T_a\). There are three possible scenarios:
1. \(T_a\) is an ancestor of \(T_b\): Here \(s_1 \prec s_2\) and therefore an edge cannot exist from \(s_2\) to \(s_1\), because a reference to \(T_b\) is not available at \(s_1\).
2. \(T_b\) is an ancestor of \(T_a\): Here the spawn of \(T_a\) precedes \(s_2\) because otherwise \(s_2\) will execute before \(s_1\) during a depth-first execution.
3. \(T_a\) completes execution before \(T_b\) during the depth-first execution: In this case \(s_1\) either precedes the spawn of \(T_b\) or \(s_1\) can execute in parallel with the
spawn of \( T_b \). If \( s_1 \) precedes the spawn of \( T_b \), a reference to \( T_b \) is not available during the execution of \( s_1 \). If \( s_1 \) can execute in parallel with the spawn of \( T_b \), then performing a get() operation on \( T_b \) by \( s_1 \) will cause a race.

Lemma 2 proves that if the program is free of races, the steps in the computation graph form a partial order. Therefore, the program has no cyclic dependencies and is free of deadlocks.

A.3 Data Race Freedom and Functional and Structural Determinism

We say that a parallel program is \textit{functionally deterministic} if it always computes the same answer, when given the same inputs. By default, any sequential computation is expected to be deterministic with respect to its inputs; if the computation interacts with the environment (e.g., a GUI event such as a mouse click, or a system call like System.nanoTime()) then the values returned by the environment are also considered to be inputs to the computation. Further, we refer to a program as \textit{structurally deterministic} if it always computes the same computation graph, when given the same inputs. Finally, following past work [18, 10], we say that a program is \textit{determinate} if it is both functionally and structurally deterministic.

The presence of data races may lead to functional and/or structural non-determinism because a parallel program with data races can exhibit different behaviors for the same input, depending on the relative scheduling and timing of memory accesses involved in a data race. In general, the absence of data races in a parallel program is not sufficient to guarantee determinism. However, the parallel constructs that are the focus of this paper (async, finish and future) were carefully selected to ensure the following \textit{Determinism Property}:

If a parallel program is written using only async, finish and future constructs, \textit{and is guaranteed to never exhibit a data race}, then it must be determinate, i.e., both functionally and structurally deterministic.

Note that the determinism property states that all data-race-free programs written using async, finish and future constructs are guaranteed to be determinate, but it does not imply that all racy programs are non-determinate.

The execution of a race-free program containing async, finish, and future constructs is \textit{dag-consistent} [3] because 1) the computation dag does not contain any cycles and 2) a read can “see” a write only if there is some serial execution order of the dag in which the read sees the write. Such a program execution is guaranteed to be deterministic. Therefore, verifying race freedom of programs is an important step in proving the correctness of programs.

B Theoretical Results

In this section, we present a worst case complexity analysis of the algorithm and the intuition for its correctness. The following theorem presents the asymptotic complexity of the race detection algorithm in Section 4.3.
Theorem 1. Consider a program with async, finish and future constructs that executes in time $T$ on one processor, creates $f$ async tasks, $n$ future tasks, performs $n$ non-tree join edges and references $v$ shared memory locations. Algorithms 1–10 can be implemented to check this program for determinacy races in $O(T(f + 1)(n + 1)\alpha(T, a + f))$ time using $O(a + f + n + v * (f + 1))$ space.

Proof. The size of the dynamic task reachability graph is $O(a + f + n)$ which includes $O(a + f)$ for the disjoint set, interval labels and LSA information and $O(n)$ for the non-tree joins. The size of the writer shadow memory is $O(v)$ and the worst case size of the reader shared memory is $O(v * (f + 1))$.

For every shared memory access, the Precede function may be invoked $f + 1$ times in the worst case. The worst case complexity of a single invocation to Precede function is $O((n + 1)\alpha(T, a + f))$ because it may visit $n$ non-tree edges in the worst case and each time it involves a disjoint set operation. Here $\alpha$ is Tarjan’s functional inverse of Ackermann’s function which, for all practical purposes is bounded above by 4. When the input program has no future tasks and no non-tree joins, the complexity of the algorithm is same as the SP-bags algorithm, because $f$ and $n$ are zero and the reader shadow space for each shared memory location contains a maximum of one reader.

We next present the intuition as to why a single run of the race detector in Algorithms 1–10 correctly detects races in programs with async, finish and future. The following two lemmas gives the reasoning for storing one (async or future) task in the writer shadow memory and one async task (and multiple future tasks) in the reader shadow memory. Lemma 3 was presented in [15] for Cilk computation graphs and it holds for computation graphs with async, finish and future.

Lemma 3. Suppose that three steps $s_1$, $s_2$, and $s_3$ execute in order in a serial, depth-first execution of a computation graph, and suppose that $s_1 \prec s_2$ and $s_1 \parallel s_3$. Then, we have $s_2 \parallel s_3$.

Proof. Please refer to [15].

Lemma 4. Suppose that three steps $s_1$, $s_2$, and $s_3$ execute in order in a serial, depth-first execution of a computation graph, and let Task($s_1$) = $T_A$, Task($s_2$) = $T_B$, Task($s_3$) = $T_C$, where $T_A$, $T_B$ and $T_C$ are async tasks. Suppose that $s_1 \parallel s_2$ and $s_2 \parallel s_3$. Then, we have $s_1 \parallel s_3$.

Proof. Since $T_A$, $T_B$ and $T_C$ are async tasks, they can be waited upon using only finish operations and not by get() operations. Let us assume $s_1 \parallel s_2$ and $s_2 \parallel s_3$ and $s_1 \prec s_3$. Since $s_1 \prec s_3$, a finish $F$ must be executed by the task which is the lowest common ancestor of $T_A$ and $T_C$ in the spawn tree. There are two cases:

1. $T_B$ is enclosed inside $F$. In this case $F$ ensures that $s_2 \prec s_3$, which contradicts our assumption that $s_2 \parallel s_3$.
2. $T_B$ is not enclosed inside $F$. In this case $F$ ensures that $s_1 \prec s_2$, which contradicts our assumption that $s_1 \parallel s_2$.

The following Lemma provides the intuition as to why it is sufficient to represent the reachability information at task level during a depth-first execution.
Lemma 5. Consider an execution of Algorithms 1–10 on a computation graph \(G\). Suppose \(s_{A1} \prec s_{B1}\) and \(s_{B2} \prec s_{C}\) where Task\((s_{A1}) = T_{A}, \text{Task}(s_{B1}) = T_{B}, \text{Task}(s_{B2}) = T_{B}\) and Task\((s_{C}) = T_{C}\) and let \(s_{C}\) be the current step being executed and suppose that \(s_{A1}, s_{B1}\) and \(s_{B2}\) have executed before \(s_{C}\) during the depth-first execution. Then, for all completed steps \(s_{A}\) such that Task\((s_{A}) = T_{A}\), we have \(s_{A} \prec s_{C}\).

Proof. We will consider four different cases based on whether the tasks are related by ancestor-descendant relationships or not:

1. \(T_{A}\) is an ancestor of \(T_{B}\) and \(T_{B}\) is an ancestor of \(T_{C}\): In this case, the completed steps of \(T_{A}\) are the steps before the spawn of \(T_{B}\) (or an ancestor of \(T_{B}\)). All such steps must precede the steps of \(T_{C}\), due to the spawn edge from \(T_{B}\) to \(T_{C}\).

2. \(T_{A}\) is an ancestor of \(T_{B}\) and \(T_{B}\) is not an ancestor of \(T_{C}\): If \(T_{C}\) is a descendant of \(T_{A}\), then all completed steps of \(T_{A}\) must precede the spawn of \(T_{C}\). Next let us consider the case where \(T_{C}\) is not a descendant of \(T_{A}\). In this case, \(T_{A}\) has completed the execution before the execution of \(T_{C}\). Since \(T_{B}\) is not an ancestor of \(T_{C}\) and \(s_{B2} \prec s_{C}\), there must be a join operation on \(T_{B}\) along the path from \(s_{B2}\) to \(s_{C}\). Since \(T_{A}\) is an ancestor of \(T_{B}\), there must also be a join operation on \(T_{A}\) along the path from \(s_{B2}\) to \(s_{C}\) before the join on \(T_{B}\). This is based on the assumption that no data races have been detected so far during the execution of the program (See Lemma 1).

3. \(T_{A}\) is not an ancestor of \(T_{B}\) and \(T_{B}\) is an ancestor of \(T_{C}\): Since \(T_{A}\) is not an ancestor of \(T_{B}\), there must be a join edge from \(T_{A}\) along the path to \(s_{B1}\). This means that for all steps \(s_{C}\) such that Task\((s_{C}) = T_{C}, s_{A} \prec s_{C}\).

4. \(T_{A}\) is not an ancestor of \(T_{B}\) and \(T_{B}\) is not an ancestor of \(T_{C}\): In this case \(T_{A}\) must have completed execution before \(T_{B}\) and \(T_{B}\) must have completed execution before \(T_{C}\) during the depth first execution. There must be a join edge from \(T_{A}\) along the path to \(s_{B1}\) and a join edge from \(T_{B}\) along the path to \(s_{C}\). This ensures that \(s_{A} \prec s_{C}\). \(\square\)

The next lemma discuss the correctness of the \textsc{Precede} function given in Algorithm 10.

Lemma 6. Consider an execution of Algorithms 1–10 on a computation graph \(G\). \textsc{Precede}(\(T_{A}, T_{B}\)) = \text{true} during the execution of \(s_{j}\), where Task\((s_{j}) = T_{B}\) if and only if \(s_{i} \prec s_{j}, \forall s_{i}\) such that Task\((s_{i}) = T_{A}\) and \(s_{i}\) executes before \(s_{j}\) during the depth first execution of \(G\).

Proof. (\(\Rightarrow\)) We will prove this by induction. Let \(s_{i} = v_{1},..v_{k},..v_{n} = s_{j}\) be a path from \(s_{i}\) to \(s_{j}\) in the computation graph. Let \(n = 1\) in which case there is an edge \((s_{i}, s_{j})\) in the computation graph.

1. \((s_{i}, s_{j})\) is a continue edge: In this case \(s_{i}\) and \(s_{j}\) belong to the same task and are represented in the dynamic task reachability graph by the same node.

2. \((s_{i}, s_{j})\) is a spawn edge: In this case \(T_{A}\) is the parent of \(T_{B}\) in the spawn tree and therefore \(T_{B}\) will have a higher preorder value and a lower postorder value than \(T_{A}\).

3. \((s_{i}, s_{j})\) is a join edge: \(T_{A}\) and \(T_{B}\) will belong to the same disjoint set, if it is a tree join. If \((s_{i}, s_{j})\) is a non-tree join edge, then \(T_{A} \in P(T_{B})\), the non-tree predecessors of \(T_{B}\).
In all three cases $\text{Precede}(T_A, T_B) = \text{true}$. Now let us assume that if $s_i \prec s_j$ with a path length of $n = k$, then $\text{Precede}(T_A, T_B) = \text{true}$. Now consider the case where $s_i \prec s_j$ with a path length of $n = k + 1$. Let $(s_i, s_j)$ be the last edge along the path from $s_i$ to $s_j$. Consider the following cases:

1. $(s_i, s_j)$ is a continue edge: In this case there is a path of length $k$ from $s_i$ and $s_j$, where $\text{Task}(s_j) = T_B$. Therefore, by our induction hypothesis $\text{Precede}(T_A, T_B) = \text{true}$. 

2. $(s_i, s_j)$ is a spawn edge: If $T_A$ is an ancestor of $T_B$ in the dynamic task reachability graph, then $T_B$ will have a higher preorder value and a lower postorder value than $T_A$ and therefore $\text{Precede}(T_A, T_B) = \text{true}$. If $T_A$ is not an ancestor of $T_B$, then there is a path from $T_A$ to $T_C$ and a path from $T_C$ to $T_B$ both of length less than or equal to $k$, where $T_C$ is an ancestor of $T_B$. There are two possible cases: 1) $T_C$ is the LSA of $T_B$, in which case $\text{Precede}(T_A, T_C) = \text{true}$ by our inductive hypothesis and 2) $T_A$ and $T_C$ are in the same disjoint set, in which case the label of $T_A$ subsumes the label of $T_B$.

3. $(s_i, s_j)$ is a join edge: If $(s_i, s_j)$ is a tree join edge, $T_B$ and $T_C$ are in the same disjoint set, where $\text{Task}(s_i) = T_C$ and $\text{Precede}(T_A, T_C) = \text{true}$ according to our inductive hypothesis. If $(s_i, s_j)$ is a non-tree join edge, then $T_A \in P(T_B)$, the non-tree predecessors of $T_B$.

$(\Rightarrow)$ Since $\text{Precede}(T_A, T_B)$ returned true, there must be a sequence of calls to $\text{Visit}$ with arguments $(T_A, T_B), (T_A, T_{X_1}), \ldots, (T_A, T_{X_n})$ where $(T_A, T_{X_k})$ returned true. Here, $T_A$ and $T_{X_k}$ must be in the same disjoint set or the label of $T_A$ must subsume the label of $T_{X_n}$. $(T_A, T_{X_{i+1}})$ is invoked from $(T_A, T_{X_k})$ if and only if $T_{X_{i+1}}$ is an immediate predecessor of the LSA of $T_{X_k}$ or if $T_{X_{i+1}}$ is an immediate predecessor of $T_{X_k}$. Therefore according to Lemma 5, $s_i \prec s_j$. 

**Theorem 2.** Algorithms 1–10 detect a determinacy race in the input program if and only if a determinacy race exists.

**Proof.** $(\Rightarrow)$ Suppose Algorithms 1–10 detect a determinacy race when executing a step $s_2$. The three possible cases are:

1. $s_2$ performs a write and $\text{Precede}(T_A, T_B) = \text{false}$, where $\text{Task}(s_2) = T_B$ and $T_A \in \text{readers}(l)$
2. $s_2$ performs a write and $\text{Precede}(T_A, T_B) = \text{false}$, where $\text{Task}(s_2) = T_B$ and $\text{writer}(l) = T_A$
3. $s_2$ performs a read and $\text{Precede}(T_A, T_B) = \text{false}$, where $\text{Task}(s_2) = T_B$ and $\text{writer}(l) = T_A$

In the first case, $T_A$ is added to $\text{readers}(l)$ by step $s_1$ when $s_1$ performed a read of $l$ (where $\text{Task}(s_1) = T_A$). Here $s_1$ executes before $s_2$ during the depth first execution. Since $\text{Precede}(T_A, T_B) = \text{false}$, $s_1$ and $s_2$ can logically execute in parallel according to Lemma 6 and therefore a determinacy race exists. The other two cases are similar.

$(\Leftarrow)$ We now show that if a program contains a determinacy race on a location $l$, our race detection algorithm reports a determinacy race on location $l$. Let $s_1$ and $s_2$ be two steps involved in a determinacy race on location $l$, where if there are multiple races on $l$, we choose the determinacy race whose second step executes earliest in the depth-first execution order of the program. There are three possible ways the determinacy race could occur:
1. $s_1$ writes $l$ and $s_2$ reads $l$: Suppose $writer(l) = T$ when $s_2$ is executed. If $T = T_A$ where $Task(s_1) = T_A$, then since $s_1 \parallel s_2$, according to Lemma 6 $\text{Precede}(T_A, T_B) = \text{false}$ and the algorithm will report a race. If $T \neq T_A$, then $writer(l)$ was last updated by step $s$ such that $s$ executed after $s_1$ but before $s_2$. If $s_1 \parallel s$, there must exist a write-write determinacy race between $s_1$ and $s$ since they access a common memory location $l$ in parallel. This contradicts our assumption that $s_2$ executes earliest during a depth-first execution among all second steps that are involved in a determinacy race on $l$. If $s_1 \prec s$, then we have have $s \parallel s_2$ by Lemma 3. According to Lemma 6 $\text{Precede}(T_A, T_B) = \text{false}$ and the algorithm will report a race.

2. $s_1$ writes $l$ and $s_2$ writes $l$: This case is similar to write-read race.

3. $s_1$ reads $l$ and $s_2$ writes $l$: Suppose $T_A \in \text{readers}(l)$, where $Task(s_1) = T_A$ when $s_2$ is executed. Then since $s_1 \parallel s_2$, according to Lemma 6 $\text{Precede}(T_A, T_B) = \text{false}$ and the algorithm will report a race. Now let us consider the case $T_A \notin \text{readers}(l)$. If $s_1$ adds $T_A$ to $\text{readers}(l)$, then $T_A$ was removed from $\text{readers}(l)$ by a step $s$ such that $s_1 \prec s$ and before $s_2$ during the depth-first execution. This implies that $s_1 \prec s$. Let us assume that there is a sequence of steps $s'_1, s'_2, \ldots, s'_n$ which reads $l$ before $s_2$ such that $s'_1 \prec s'_2 \prec \ldots \prec s'_n$, where $s = s'_1$. By transitivity, we have $s_1 \prec s'_n$. By Lemma 3, it follows that $s'_n \parallel s_2$, since $s_1 \parallel s_2$ and therefore a race between $s'_n$ and $s_2$ will be reported. Now let us consider the case where $s_1$ does not add $T_A$ to $\text{readers}(l)$. In this case, $T_A$ must be an async task and during the execution of $s_1$, $\text{readers}(l)$ contains $T'$, where $Task(s') = T'$. Here $s'$ performed a read of $l$ and $s' \parallel s_1$ and $T'$ also must be an async task. Since $s_1 \parallel s_2$, by Lemma 4 $s' \parallel s_2$. Looking at the sequence of updates to $\text{readers}(l)$, there must exist a task $T_K \in \text{readers}(l)$, where $Task(s_k) = T_K$, $s_k$ performed a read of $l$ and $s' \prec s_k$. Since $s' \parallel s_2$, Lemma 3 implies that $s_k \parallel s_2$ and a race between $s_k$ and $s_2$ will be reported. 