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, constructed using 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 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 wait for a child task [16, 4], a descendant task [26, 27] or the immediate left sibling [14]. 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. Both approaches can lead to non-strict computation graphs, in general. Race detection algorithms based on vector clocks [3, 17] 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 [16, 4, 26, 27] focus on an imperative 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 [10]. The addition of point-to-point synchronization with 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 [10] and Habanero-Java [8], spawn-sync constructs in Cilk [5], and task-taskwait constructs in OpenMP [24].

Existing algorithms for detecting determinacy races for dynamic task parallelism, do not support race detection for futures. For instance, data race detectors for Cilk [16, 4] handle only spawn-sync constructs where the computation graph is a Series-Parallel (SP) dag. Although the computation graphs for async-finish parallelism [26, 27] 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 [26, 27]. 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 [19, 12]. Thus, a data race detector for programs with async, finish, and future constructs, can be used as a determinacy checker for these programs.

The main contributions of this paper[1] 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 algorithms, 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 async-finish parallelism and point-to-point synchronization using futures. 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 determinacy races 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.

[1] A summary abstract of this approach was presented as a brief announcement at SPAA 2016 [28].

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 [10], Habanero Java [8], Chapel [9], 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 dynamic 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 [18] (or promise [21]) 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 = async<T> 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 on 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.

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* [6], and the class of computations generated by async-finish constructs is called *terminally strict* [1].

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. This example has a non-strict computation graph, because of the get operations performed by T_B and T_C on their siblings.

```
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  Stmt8;
9  A.get();
10 Stmt9;
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* [6] 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 sequence of instruction instances contained in a task such that no 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 synchronization among tasks. When task A performs a get on future B, a join edge (also referred to as a "future join edge") is inserted from the last step of B to the step in task A that immediately follows the `get()` operation. In addition, "finish 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 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*. Note that all finish join edges must be tree joins, and some future join edges may be tree edges and some may be non tree edges.

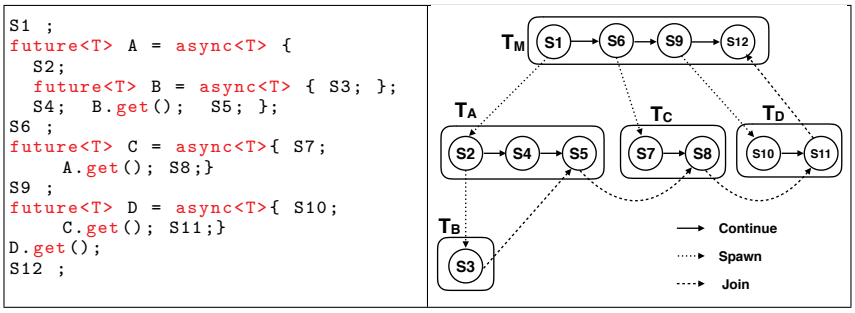

Fig. 2: Example program with futures and its computation graph. $S1$-$S12$ are steps in the program.The circles represent the steps in the program. The rectangles represents tasks. T_M is the main task and T_A, T_B, T_C and T_D are future tasks created during the execution of the program.

All three kinds of edges have been studied in past work on computation graphs for the Cilk [5] and Habanero-Java [26] 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.*

This precedence relation is a partial order, and is also referred to as the "happens-before" relation in past work [20]. 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 may execute in parallel, denoted $u \parallel v$, iff $u \not\prec v$ and $v \not\prec u$.

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

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 $S2 \parallel S10$ because there is no directed path from $S2$ to $S10$, or from $S10$ to $S2$, 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 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 [19, 12], 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 (presented in Section 4.2) to answer the second question. Section 4.3 presents our overall determinacy race detection algorithm.

4.1 Dynamic Task Reachability Graph

Since storing the entire computation graph of the program execution is usually intractable due to memory limitations (akin to storing a complete dynamic trace of a program), we introduce a more compact representation that still retains 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. The representation assumes that the input program is executed serially in depth-first order, and leverages 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 [11, Chapter 22], which maintains a dynamic collection of disjoint sets Σ and provides three operations:

1. MakeSet(x) which creates a new set that contains x and adds it to Σ
2. Union(X, Y) which performs a set union of X and Y, adds the resulting set to Σ and destroys set X and Y
3. 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 [30]. Here α is 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 [13], 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 correspond to 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 $[x.pre, x.post]$

is the interval associated with task x and $[y.pre, y.post]$ is the interval associated with task y, then x is an ancestor of y if and only if $x.pre \leq y.pre$ and $y.post \leq x.post$. When task A performs a join operation on a descendant task B, 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 [13] 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 in the spawn tree that have performed at least one non-tree join operation. For each task, we store only the lowest significant ancestor.

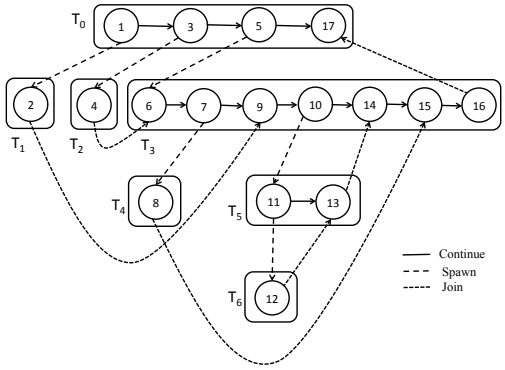

Fig. 3: A computation graph with non-tree joins. The join edges (2,9) and (4,6) are non-tree joins because T_1 and T_2 are not descendants of T_3.

Definition 4. *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 dynamic task instance.*
- *$D = \{D_i\}_{i=1}^{n}$ 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.*

– $P : N \rightarrow 2^N$ *represents the set of non-tree edges* $P(u) = \{v_1, .., v_k\}$ *if and only if there are non-tree join edges from tasks* $v_1..v_k$ *to* u.
– $A : N \rightarrow N$ *represents the lowest ancestor with at least one incoming non-tree edge.* $A(u) = v$, *if and only if* $w_1, w_2..w_k..w_m$ *(where* $r = w_1$, $v = w_k$ *and* $u = w_m$*) is the path consisting of spawn edges from the root* r *of* G *to* u, *and* $P(w_j) = \emptyset, \forall j$ *such that* $k + 1 \leq j \leq m - 1$ *and* $P(v) \neq \emptyset$. v *is referred to as the* lowest significant ancestor (LSA) *of* u.

Disjoint Set	Task	L (Label)	P (NT)	A (LSA)
0	T_0	[0, MAXINT]	( )	-
1	T_1	[1, 2]	( )	-
2	T_2	[3, 4]	( )	-
3	T_3	[5, MAXINT-1]	{T_1,T_2}	-
4	T_4	[6, 7]	( )	T_3
5	T_5	[8, MAXINT-2]	( )	T_3
6	T_6	[9, MAXINT-3]	( )	T_3

(a)

Disjoint Set	Task	L (Label)	P (NT)	A (LSA)
0	T_0	[0, 13]	{T_1, T_2}	-
	T_3	[5, 12]		
	T_4	[6, 7]		
	T_5	[8, 11]		
	T_6	[9, 10]		
1	T_1	[1, 2]	( )	-
2	T_2	[3, 4]	( )	-

(b)

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.

Table 1(a) shows the dynamic task reachability graph for the computation graph in Figure 3 after the execution of step 11. Here the postorder values assigned to T_0, T_3, T_5 and T_6 are temporary values (See Section 4.3). All tasks are in a separate disjoint sets, because no tree joins have been performed yet. Table 1(b) shows the dynamic task reachability graph for the computation graph in Figure 3 after the execution of step 17.

4.2 Shadow Memory

As in past work [26, 27], 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 *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 occurs: task creation, task return, begin-finish, end-finish, `get()` operation, shared

```
Input: Main task M
1: dfid ← 0
2: tmpid ← MAXINT
3: S_M ← MAKE-SET(M)
4: S_M.pre ← dfid
5: dfid ← dfid + 1
6: S_M.post ← tmpid
7: tmpid ← tmpid − 1
8: S_M.parent ← null
9: S_M.lsa ← null
```

Algorithm 1: Initialization

```
Input: Parent task P, Child task C
1: S_C ← MAKE-SET(C)
2: S_C.pre ← dfid; dfid ← dfid + 1
3: S_C.post ← tmpid; tmpid ← tmpid − 1
4: S_C.parent ← S_P
5: if S_P.nt = {} then
6:     S_C.lsa ← S_P.lsa
7: else
8:     S_C.lsa ← S_P
9: end if
```

Algorithm 2: Task creation

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 $[0, MAXINT]$ 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 shows the actions performed by the race detector at the start and end of a finish. At the end of a finish F, the disjoint sets of all tasks with F as the immediately enclosing finish is merged with the disjoint set of the ancestor task executing the finish.

Input: Terminating task C
1: $S_C.post \leftarrow dfid$; $dfid \leftarrow dfid + 1$
2: $tmpid \leftarrow tmpid + 1$

Algorithm 3: Task termination

Input: Tasks A, B such that A performs B.`get`()
1: **if** FIND-SET(A) =
2: FIND-SET($B.parent$) **then**
3: MERGE(S_A, S_B)
4: **else**
5: $S_A.nt \leftarrow S_A.nt \cup \{B\}$
6: **end if**

Algorithm 4: Get operation

Input: Start of finish F in task A
1: $F.parent \leftarrow A$

Algorithm 5: Start finish

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

Algorithm 6: End finish

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

Algorithm 7: Merge tasks

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, 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, PRECEDE routine shown in Algorithm 10 checks if task A must precede B by invoking routine VISIT which is also given in Algorithm 10. Lines 6–11 of 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 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

```
Input: Memory location M, Task A that
    writes to M
 1: for X ∈ M_s.r do
 2:     if not PRECEDE(X, A) then
 3:         a determinacy race exists
 4:     else
 5:         M_s.r ← M_s.r − {X}
 6:     end if
 7: end for
 8: if not PRECEDE(M_s.w, A) then
 9:     a determinacy race exists
10: end if
11: M_s.w ← A
```

Algorithm 8: Write check

```
Input: Memory location M, Task A that
    reads M
 1: update = false
 2: for X ∈ M_s.r do
 3:     if PRECEDE(X, A) then
 4:         M_s.r ← M_s.r − {X}
 5:         update ← true
 6:     else if ISFUTURE(X) or
 7:             ISFUTURE(A) then
 8:         update ← true
 9:     end if
10: end for
11: if not PRECEDE(M_s.w, A) then
12:     a determinacy race exists
13: end if
14: if update then
15:     M_s.r ← M_s.r ∪ {A}
16: end if
```

Algorithm 9: Read check

```
Input: Tasks A, B
 1: procedure PRECEDE(A, B)
 2:     return VISIT(A, B, {})
 3: end procedure

 1: procedure VISIT(A, B, Visited)
 2:     if B ∈ Visited then
 3:         return false
 4:     end if
 5:     Visited ← Visited ∪ {B}
 6:     S_A ← FIND-SET(A)
 7:     S_B ← FIND-SET(B)
 8:     if S_A.pre ≤ S_B.pre and
 9:        S_A.post ≥ S_B.post then
10:         return true
11:     end if
12:     if S_A.pre > S_B.pre then
13:         return false
14:     end if
15:     for all x in S_B.nt do
16:         if VISIT(A, x, Visited)
17:         then
18:             return true
19:         end if
20:     end for
21:     sa ← B.lsa
22:     while sa ≠ null do
23:         for all x in sa.nt do
24:             if VISIT(A, x,
                    Visited) then
25:                 return true
26:             end if
27:         end for
28:         sa ← sa.lsa
29:     end while
30:     return false
31: end procedure
```

Algorithm 10: Reachability check

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 algorithm. The proofs for these theorems are given in [29].

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 $O(T(f+1)(n+1)\alpha(T, a+f))$ time using $O(a+f+n+v*(f+1))$ space.*

Here α is functional inverse of Ackermann's function which, for all practical purposes is bounded above by 4. It is interesting to note that our algorithm

degenerates to past complexity results for async-finish programs [26] in the case when the program creates no futures ($f = n = 0$).

Theorem 2. *Algorithms 1–10 detect a determinacy race for a given parallel program and data input 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) [23]. The PIR extends Soot's Jimple IR [31] 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.

Benchmark	**#Tasks**	**#NTJoins**	**#SharedMem**	**#AvgReaders**	**Seq (millisecs)**	**Racedet (millisecs)**	**Slowdown (Racedet/Seq)**
Series-af	999,999	0	4,000,059	0.75	483,224	484,746	1.00
Series-future	999,999	0	6,000,059	0.66	487,134	487,985	1.00
Crypt-af	12,500,000	0	1,150,000,682	0.74	15,375	119,504	7.77
Crypt-future	12,500,000	0	1,175,000,682	1.23	15,517	128,234	8.26
Jacobi	8,192	34,944	641,499,805	1.70	3,402	27,388	8.05
Strassen	30,811	33,612	1,610,522,196	0.94	6,281	33,618	5.35
Smith-Waterman	1,608	4,641	1,652,175,806	1.56	3,488	34,558	9.92

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 [7] 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 2048×2048 matrix, where each tasks computes a 64×64 submatrix.
- **Strassen:** Multiplication of 1024×1024 matrices using Strassen's algorithm. The implementation uses a recursive cutoff of 32×32.
- **Smith-Waterman:** Sequence alignment of two sequences of size 10000. The alignment matrix computation is done by 40×40 future tasks.

The first four benchmarks were derived from the original versions in the JGF suite. The next two, Jacobi and Strassen were translated by the authors from OpenMP versions of those programs in the Kastors [32] 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 ≥ 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 $\approxeq 100\times$ 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 [25] 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 due to 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 5.35×) 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 in the graph traversal.

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., [3, 17]. Atzeni et al. [2] presented a low overhead, high accuracy vector clock race detector for OpenMP programs via a combination of static and dynamic analysis. 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 [22] presented the Offset-Span labeling algorithm for nested fork-join constructs, which is an extension of English-Hebrew labeling scheme [15]. 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 dynamic fork-join nesting depth in the program. Our approach uses a constant size labeling scheme to store reachability information for ancestor-descendant tasks. While the Offset-Span labeling algorithm supports only nested fork-join constructs, our algorithm supports a more general set of computation graphs.

Feng and Leiserson [16] 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. [4] presented the 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 include an implementation of the algorithm. Raman et al. [26] extended the SP-bags algorithm to support async-finish parallelism. They subsequently proposed the parallel SPD3 algorithm [27] also for async-finish constructs. 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 parallel executions with that input are guaranteed to be race-free and deterministic. 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, discussed its correctness, and evaluated an implementation of the algorithm on a range of benchmarks that generate both strict and non-strict computation graphs. 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 $1.00\times$ – $9.92\times$, which is very much in line with slowdowns experienced for fully strict computation graphs.

References

1. Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA '07*, pages 229–240, New York, NY, USA, 2007. ACM.
2. Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Gregory L Lee, Ignacio Laguna, Martin Schulz, Joachim Protze, and Matthias Mueller. Archer: Effectively spotting data races in large openmp applications. In *IPDPS'16*, pages 53 – 62, May 2016.
3. Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *PADTAD '06*, pages 69–78, New York, NY, USA, 2006. ACM.
4. Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *SPAA '04*, pages 133–144, New York, NY, USA, 2004. ACM.
5. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP '95*, pages 207–216, New York, NY, USA, 1995. ACM.
6. Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.
7. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.
8. Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *PPPJ '11*, pages 51–61, New York, NY, USA, 2011. ACM.
9. B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.
10. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.
11. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
12. Jack B. Dennis, Guang R. Gao, and Vivek Sarkar. Determinacy and repeatability of parallel program schemata. In *DFM '12*, pages 1–9, Washington, DC, USA, 2012. IEEE Computer Society.
13. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC '87*, pages 365–372, New York, NY, USA, 1987. ACM.
14. Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. Race detection in two dimensions. In *SPAA '15*, pages 101–110, New York, NY, USA, 2015. ACM.
15. A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPOPP '90*, pages 1–10, New York, NY, USA, 1990. ACM.
16. Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA '97*, pages 1–11, New York, NY, USA, 1997. ACM.
17. Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09*, pages 121–133, New York, NY, USA, 2009. ACM.
18. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
19. Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, May 1969.

20. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
21. B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88*, pages 260–267, New York, NY, USA, 1988. ACM.
22. John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91*, pages 24–33, New York, NY, USA, 1991. ACM.
23. V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(1), April 2013.
24. OpenMP specifications. http://www.openmp.org/specs.
25. Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *RV'10*, pages 368–383, Berlin, Heidelberg, 2010. Springer-Verlag.
26. Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. *Formal Methods in System Design*, 41(3):321–347, December 2012.
27. Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *PLDI '12*, pages 531–542, New York, NY, USA, 2012. ACM.
28. Rishi Surendran and Vivek Sarkar. Brief announcement: Dynamic determinacy race detection for task parallelism with futures. In *SPAA'16*, Pacific Grove, CA, USA, July 2016.
29. Rishi Surendran and Vivek Sarkar. Dynamic determinacy race detection for task parallelism with futures. Technical Report TR16-01, Department of Computer Science, Rice University, Houston, TX, 2016.
30. Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.
31. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99*. IBM Press, 1999.
32. Philippe Virouleau, Pierrick Brunet, Franois Broquedis, Nathalie Furmento, Samuel Thibãult, Olivier Aumage, and Thierry Gautier. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *IWOMP 2014*, pages 16–29, 2014.