Innovations in Systems and Software Engineering (2019) 15:289–306
https://doi.org/10.1007/s11334-019-00343-5

S.I. : NFM2018

Model-checking task-parallel programs for data-race

Radha Nakade[1] · Eric Mercer[1] · Peter Aldous[1] · Kyle Storey[1] · Benjamin Ogles[1] · Joshua Hooker[1] · Sheridan Jacob Powell[1] · Jay McCarthy[2]

Received: 3 October 2018 / Accepted: 8 May 2019 / Published online: 18 May 2019

Abstract
Many of the correctness properties afforded by task-parallel programming models such as OpenMP, Cilk, X10, Chapel, Habanero, etc. rely on data-race freedom. The research in this paper studies data-race in the context of these models with the intent to prove with model checking its absence on any feasible schedule for a given input. The paper presents the computation graph as a representation of a happens-before relation that additionally tracks memory accesses with a quadratic algorithm to detect data-race on the graph. It then shows how the graph is constructed from an execution of a task-parallel program and proves that under a fixed order of mutual exclusion, if a schedule with a data-race exists in the program, then a data-race is manifest in the computation graph. The paper then defines a model checking algorithm that enumerates all orders of mutual exclusion to prove data-race freedom over all schedules on the given input. The approach is evaluated in a Java implementation of Habanero using the JavaPathfinder model checker. The results, when compared to other data-race detectors including one based on vector clocks, show that this new approach is more efficient than existing JavaPathfinder solutions and is comparable to the vector clock solution in the absence of data-race but slower in the presence of data-race since the vector clock algorithm is on-the-fly while the new approach is not. The results also show that the new approach avoids the memory overhead of vector clocks when there are many tasks and objects to track.

Keywords Model checking · Data-race · Task parallel · Cilk · x10 · Habanero · OpenMP · SP-bags · Vector clocks · Formal verification · Happens-before · Partial order · Computation graph · Determinism

1 Introduction

A *data-race* is where two concurrent executions access the same memory location with at least one of the two accesses being a write. It introduces non-determinism into the program execution as the behavior may depend on the order in which the concurrent executions access memory. Data-race is problematic because it is not possible to directly control or observe the run-time internals to know if a data-race exists, let alone enumerate program behaviors when one does.

The *data-race detection* problem, given a program with its input, is to determine if there exists an execution containing a data-race. The research presented in this paper is concerned with proving data-race freedom for *task-parallel models* that impose structure on parallelism by constraining how threads are created and joined, and by constraining how shared memory is accessed (e.g., OpenMP, Cilk, X10, Chapel, Habanero, etc.). These models rely on run-time environments to implement task abstractions to represent concurrent executions [4,7,8,21]. The language restrictions on parallelism and shared memory interactions enable properties like *determinism* (i.e., the computation is independent of the execution) or the ability to *serialize* (i.e., removing all task related keywords yields a serial solution). Such properties only hold in the absence of data-race, which is not always the case since programmers, both intentionally and unintentionally, move outside the programming model.

This work is supported by the National Science Foundation under Grant 1302524.

✉ Eric Mercer
egm@cs.byu.edu

Radha Nakade
radha.nakade@gmail.com

Peter Aldous
aldous@cs.byu.edu

Jay McCarthy
jay.mccarthy@gmail.com

[1] Brigham Young University, Provo, Utah, USA

[2] University of Massachusetts Lowell, Lowell, USA

Data-race detection in task-parallel models generally prioritizes performance and the ability to scale to many tasks over a proof of absence. The predominant *SP-bags* algorithm, with its variants, is a dynamic approach that exploits assumptions on task creation and joining for efficient on-the-fly detection with low overhead [2,9,15,33,36]; millions of tasks are feasible with varying degrees of slow-down (i.e., slow-down increases as parallelism constraints are relaxed) [34,35]. The approaches, and SP-bags in general with all its variants, do not work in the presence of mutual exclusion. Other approaches use access histories [30,32], access sets [28], or programmer annotations [40]. Performance is a priority requiring careful integration into complex run-time environments, and solutions are often only *complete*, meaning that little can be concluded about other executions of the program on the same input.

The research presented in this paper reprises the data-race problem in task-parallel models with the intent to prove, via model checking, data-race freedom on a given input over all feasible executions with support for mutual exclusion. Prior model checking-based solutions enumerate schedules that interleave conflicting accesses, meaning at least one access is a write, to shared variables [1,17,42]. In this way, the model checker schedules on every shared variable access to enumerate all interesting schedules on which a data-race might exist, but such an approach comes with an exponential cost in the number of schedules generated; these solutions quickly run out of resources even on very small toy examples on programs that are data-race free. For input programs with data-race, performance depends on the luck of the scheduler to choose a schedule on which the data-race manifests before running out of resources.

The approach here rather uses techniques from dynamic approaches to build a happens-before relation in the form of a computation graph from a single observed program execution sufficient to prove data-race freedom in all executions that order mutually exclusive regions in the same way as the observed execution [20,24,27]. The happens-before relation is thus a partial order and an equivalence relation. The observed trace being representative of all other schedules that observe the same order on the mutually exclusive regions but perhaps order other concurrent actions differently.

Unlike dynamic approaches, though, the approach here further generates other program executions necessary to prove data-race freedom over all executions on the input. As a result, in the absence of mutual exclusion, a single program execution is sufficient to prove data-race freedom. In the presence of mutual exclusion, the model checker generates and checks all feasible orderings of the mutually exclusive regions to complete the proof. Underlying this contribution is the fact that we assume the program under test terminates; if such is not the case, then the research in this paper does not directly apply.

The research presented in this paper includes an empirical study of the proposed model checking algorithm for a Java implementation of Habenero with the Java Pathfinder model checker (JPF). Unlike prior solutions, this implementation uses an idealized verification run-time for Habanero rather than a production run-time, does not require internal modifications to JPF, and gives results about the input program that generalize to any language run-time implementation [1,17,42]. Results over several published benchmarks comparing to JPF's default race detection using partial order reduction and a task-parallel approach with permission regions show the approach here to be more efficient in JPF terms with its inherent overhead.

An additional algorithm using vector clocks is also implemented in JPF, [16], as part of the research presented in this paper and compared to the proposed model checking algorithm here. This comparison shows that in the absence of data-race, the two approaches are comparable in terms of time and resources. In the presence of data-race, the vector clock algorithm, being on-the-fly, typically finds the data-race and terminates before the approach min this paper that has to wait until the execution completes before doing the data-race analysis. That said, the results also show that the vector clock analysis runs out of memory for examples with many tasks and objects to track.

Of course, as with any model checking approach, the intent is to not scale to millions of parallel tasks with hundreds of mutually exclusive regions; rather, this research assumes that it is possible to provide input to any given program that results in hundreds of tasks and a few mutually exclusive regions. It further assumes that a data-race freedom proof on the small instance of the program, the one that results in hundreds of tasks and a few mutually exclusive regions, generalizes to a large instance of the same program with millions of tasks and many mutually exclusive regions. In other words, without changing the program text in any meaningful way, it is possible to give test input appropriate for model checking. The primary contributions are thus

- a simple approach to data-race detection in programs that terminate based on creating a happens-before relation from an execution of a task-parallel program;
- a proof that scheduling to interleave mutually exclusive regions is sufficient to prove data-race freedom; and
- an implementation of the approach for Java Habanero in JPF with an implementation of a algorithm that uses vector clocks both with results from benchmarks comparing to other solutions in JPF.

The rest of this paper is organized as follows: Sect. 2 illustrates the approach in a small example; Sect. 3 defines the computation graph and an algorithm to detect data-race on a computation graph; Sect. 4 defines the language and seman-

tics of parallel tasks and shows how to build a computation graph from a program execution; Sect. 5 gives a correctness proof; Sect. 6 presents the model checking algorithm and proves it generates all interesting schedules over mutual exclusion; Sect. 7 is the empirical study with a summary of the implementation; Sect. 8 briefly discusses related work; and Sect. 9 is the conclusion with future work.

2 Example

The approach to data-race detection in this paper is presented in a very simple example. Consider the task-parallel program in Fig. 1a. The language used is defined in this paper with a formal semantics to facilitate proofs but has a direct expression in most task-parallel languages. For example, Fig. 1b shows the equivalent program in the Habanero Java language.

For Fig. 1, execution begins with the procedure `m`. The variable `g` is global. The **async** statement creates a new asynchronous task running procedure `p` passing 0 for its parameter. The task handle is stored in the region r, also global, and when that task completes and joins with its parent `m`, it runs a default λ-expression as a return-value handler. In this case, that handler is defined to be the no-op `skip`.

The **isolated** statement runs the statement in its scope in mutual exclusion to other **isolated** statements. The **await** statement joins all tasks in region `r` with the task that issued the await. The issuer may join with a task in the region if that task is has evaluated the expression in its **return** statement. That value, at the join, is passed to the return-value handler in the parent context. The parent blocks at the **await** statement until it has joined with all tasks in the indicated region.

The program in Fig. 1 has a schedule dependent data-race. If the scheduler runs the **isolated** statement in procedure `p` before the **isolated** statement in procedure `m` then there is a write–write data-race; otherwise, there is no data-race.

Related work in model checking task-parallel languages enumerates schedules to interleave the mutual exclusion and to interleave unprotected shared memory access leading quickly to state explosion [1,17,42]. These approaches use the happens-before relation to detect data-race but not to reduce the number of considered schedules—every schedule is checked.

The approach in this paper exploits so-called partial order analyses to reduce the number of schedules that must be checked to prove data-race freedom. The approach uses the simple happens-before partial order, [27], but is easily extended to something like weak causally precedes to further reduce checked schedules [20,24]. Unlike other partial-order approaches though, a sufficient set of schedules is checked to prove data-race freedom on the given input.

The approach dynamically detects shared memory accesses and uses the language semantics to capture the happens-before relation in the form of a computation graph during execution. Figure 2b shows the computation graph for the data-race free schedule of the example program. Every node represents a block of sequential operations and edges order the nodes. The thick `a`-labeled line is the result of the **async** statement creating a new task, and the dashed boxes are the **isolated** statements. Intuitively, the computation graph is a Hasse diagram with inverted edges—things at the bottom happen-after things at the top—and with extra information on each node to indicate read and write memory locations. For example, the $\omega(g)$ label indicates that variable g is written in the node. Such a graph can be checked for data-race using any number of algorithms [16,27], or if there is no isolation [22,28]. This research uses as simpler algorithm with worse complexity but supports all the language features and is easy to reason about in the proof.

To reason over all schedules, the approach in this paper assumes input programs are well formed and terminate. Well formed in this context captures common properties of task-parallel programming models that prohibit deadlock and non-determinism in how tasks are synchronized. Under these restrictions, the model checker, to prove data-race freedom, must generate a set of schedules that contains all ways

Fig. 1 A program with data-race. **a** Task-parallel. **b** Habanero Java

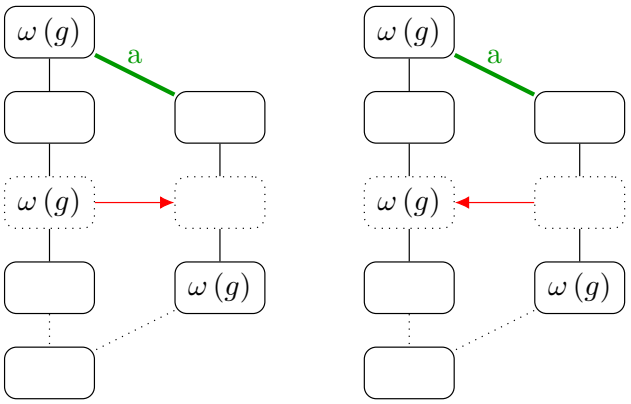
```
proc m (var x : int)
  g := 0;
  async ← p 0;
  [ isolated g := 1 ]
  await r
  return x

proc p (var x : int)
  [ isolated skip; ]
  g := 2;
  return 0
```

(a)

```
public class Example1{
  static int g = 0;
  public static void main(String[] args) {
    g := 0
    finish {
      async { p(0); }
      isolated{ g := 1; }
    }
    public static void p(int x) {
      isolated{ /* skip */ };
      g := 2;
      return 0;
    }
  }
```

(b)



(a) No data - race **(b)** Data - race

Fig. 2 Two computation graphs for Fig. 1

allowed by the program semantics to interleave **isolated** statements. Such an algorithm is presented with a proof that it enumerates all such schedules.

Figure 2b shows the computation graph for the data-race schedule of the simple example program. Although the two schedules in Fig. 2 are the only schedules that need to be considered by the model checker in this example, the number of interesting schedules grows exponentially in the number of concurrent dependent **isolated** statements. The growth limits the model-checking approach in this paper to input programs that can be instantiated in such a way as to not require resources beyond the limits of the model checker. In other words, a program intended to run on inputs that create a very large number of tasks and mutually exclusive regions has a test mode that does not meaningfully change the program text and only creates a modest number of tasks and mutually exclusive regions. A data-race proof in the test mode would then generalize to a full-scale input since the program text and underlying concurrent structure is unchanged.

3 Data-race detection on computation graphs

These sections define the computation graph with an algorithm to detect data-race given a computation graph as input. A complexity proof is then given showing the algorithm to be quadratic when the number of heap accesses in any given node is small relative to the number of nodes in the graph and otherwise cubic. The relation between the algorithm here and the SP-bags algorithm is discussed. The algorithm is then proved to be sound and complete.

3.1 Computation graphs

A computation graph for a task-parallel program is a directed acyclic graph representing the concurrent structure of the program execution [11]. It is modified here to track memory locations accessed by tasks. For the definition, `Globals` is the set of the unique identifiers for the shared locations, and $\mathcal{P}$ (`Globals`) is the power set.

Definition 1 A computation graph is defined as a directed acyclic graph (DAG), $G = (N, E, \rho, \omega)$, where

- N is a finite set of nodes;
- $E \subseteq N \times N$ is a set of directed edges;
- $\rho : (N \mapsto \mathcal{P}(\texttt{Globals}))$ maps N to the unique identifiers for the shared locations read by the tasks; and
- $\omega : (N \mapsto \mathcal{P}(\texttt{Globals}))$ maps N to the unique identifiers for the shared locations written by the tasks.

The graph collapses sequential accesses to memory locations into single nodes between concurrent operations to create, join, or isolate concurrent executions. The structure of the graph is the happens-before relation ordering shared accesses. Any feasible execution schedule of a task-parallel program can be captured in a computation graph.

3.2 Data-race detection

Let $\prec \subset N \times N$ be the happens-before relation in the graph G with constant time lookup. Such a relation can be computed in $O(|N| * |E|)$ time. There is a data-race in the graph if and only if there are two nodes, n_i and n_j, such that the nodes are concurrent, (i.e., $n_i \not\prec n_j \wedge n_j \not\prec n_i$ or, equivalently, $n_i \,||_{\prec}\, n_j$), and the two nodes conflict:

$$\text{conflict}(n_i, n_j, \rho, \omega) = \begin{array}{l} \rho(n_i) \cap \omega(n_j) \neq \emptyset \vee \\ \rho(n_j) \cap \omega(n_i) \neq \emptyset \vee \\ \omega(n_i) \cap \omega(n_j) \neq \emptyset \end{array} \quad (1)$$

The algorithm to detect data-race in a computation graph is given in Algorithm 1. It relies on the topological ordering of the nodes in the computation graph which can be computed in $O(|N| * |E|)$ time. The algorithm takes advantage of the topological sort to only check for conflicts with nodes that do not happen before the current node being considered in the sort by only looking at nodes that come after in the sort; though, this small optimization does not change the inherent complexity of the algorithm.

If there is a small bound on the number of reads and writes in any given node, meaning that that bound is much much smaller than the number of nodes in the computation graph, then the call to the *conflict* function on the inner loop is considered constant time rather than linear time (e.g., computing the intersection is negligible). As such, the data-race detection algorithm operates in quadratic or cubic time, depending on the number of reads and writes in the graph's nodes. Most often, except in the contrived benchmarks to characterize the

Algorithm 1 Data Race detection in a computation graph.

```
1: procedure DetectRace((N, E, ρ, ω))
2:     (n_0, n_1, ..., n_m) = topological_order(N, E)
3:     for all n_i ∈ (n_0, n_1, ..., n_{m-1}) do
4:         for all n_j ∈ (n_{i+1}, n_{i+2}, ..., n_m) do
5:             if (n_i ⊀ n_j) ∧ conflict(n_i, n_j, ρ, ω) then
6:                 Report data-race and exit
7:             end if
8:         end for
9:     end for
10: end procedure
```

algorithm, the bound on the reads and writes in any given node is much smaller than the number of nodes in the graph.

It is worth noting that there are more efficient algorithms to compute the data-race in the computation graph when there is no mutual exclusion, [22,28], or with mutual exclusion using vector clocks, [16,27]. Algorithm 1 is preferred here or its simplicity relative to the proofs, and more critically, the cost of data-race detection is not being driven by the algorithm to detect data-race in the detection; it is rather driven by generating the needed executions for the proof as is shown in Sect. 7.

That said, this research did implement an approach based on vector clocks, [16], and though not proven that it has the same correctness properties as Algorithm 1, assuming it does, it plugs directly into the model checking algorithm presented in this paper and all the same properties hold. The results show that since it is on-the-fly, it can get lucky and report a race before running out of resources unlike Algorithm 1.

It may be helpful to pause here to make clear the difference between an SP-bags algorithm and Algorithm 1. SP-bags relies on a special traversal of the computation graph so that least common ancestor queries combined with some book keeping determine the parallel relationship of any two nodes in the graph. As such, it is based on Tarjans fast LCA algorithm. Algorithm 1 is only leveraging the fact that the computation graph encodes the happens before relation and is not sensitive to the structure the graph or the observed execution order. This independence is also true of algorithms based on vector clocks; though, those incur some non-trivial memory overhead in the clocks as shown in Sect. 7. As such, Algorithm 1 works on computation graphs that otherwise break SP-bags and its variants.

3.3 Proof of correctness

Theorem 1 (Soundness of Algorithm 1) *If Algorithm 1 finds a data-race when applied to a computation graph G, G contains a data-race.*

Proof The condition on line 5 is the definition of a data-race in a computation graph; though the symmetric condition of $n_j \not\prec n_i$ is implied by the fact that n_i is topologically ordered before n_j by the outer loop on line 3. Algorithm 1 reports a data-race only on line 6, which is only reachable if the condition on line 5 is true. Therefore, Algorithm 1 can only report a data-race on a graph G if that data-race exists in G. □

Theorem 2 (Completeness of Algorithm 1) *If a computation graph G contains one or more instances of data-race, Algorithm 1 finds at least one instance when applied to G.*

Proof Data race is only defined over distinct nodes in G; it is impossible for a node to race with itself. The condition on line 5 is agnostic to the order in which it receives nodes; it is true for (n_i, n_j) if and only if it is true for (n_j, n_i). The proof of Theorem 1 establishes that the condition on line 5 matches the definition of data-race in a computation graph and that line 6 always reports a data-race. As such, as long as each pair of nodes is considered, a date-race is reported.

The loop defined on line 3 guarantees that every node is considered as n_i. The loop defined on line 4 guarantees that every node after n_i is considered as n_j. line 6 can break out of the loop, but only after reporting a valid data-race. In this case, the lemma holds. If no data-race is detected, every node is compared against every other node at line 5 once, line 6 never executes, and the algorithm correctly reports no data-race. □

4 Capturing computation graphs from parallel tasks

This section formally defines how to build computation graphs from parallel task executions. The semantic model for parallel tasks in this presentation is inspired by that of *isolated parallel tasks* [5]. In that model, concurrent computations are hierarchically divided into concurrent tasks with each task executing sequentially. Tasks additionally maintain *regions* that track *handles* to other tasks.

The initial task begins without any task handles. When a task t creates a child task t', it stores the handle for t' in one if its regions; t and t' are concurrent at this point and t' is able to recursively create new tasks and store handles in its own regions. If task t requires the computation of task t', then it must *await* the completion of t' and in so doing possibly block its own execution if t' is yet to complete. When t' completes, t consumes its handle. The value returned by t' is combined into the state of t through a programmer supplied *return-value handler*. Parent tasks may transfer ownership of subordinate tasks to children at creation, and children always pass unconsumed subordinate tasks to parents at completion.

Parallel tasks, unlike isolated parallel tasks, allow shared memory for communication between tasks, restrict where parents are allowed to pass tasks to children, and force the programmer to define a total order on synchronizing with

$$
\begin{aligned}
\mathbf{P} &::= (\mathbf{proc}\ p\ (\mathbf{var}\ \mathtt{l} : L)\ s)* \\
\mathbf{s} &::= s;\ s \mid \mathtt{l} := e \mid \mathbf{skip} \mid [\mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ s] \\
&\quad \mid [\mathbf{while}\ e\ \mathbf{do}\ s] \mid \mathbf{return}\ e \\
&\quad \mid \mathbf{async}\ r \leftarrow p\ e \mid \mathbf{future}\ r \leftarrow p\ e\ d\ \mathbf{r} \mid \mathbf{await}\ r \\
&\quad \mid [\mathbf{isolated}\ s] \mid \mathbf{isolated\text{-}end}
\end{aligned}
$$

Fig. 3 The surface syntax for task-parallel programs

child tasks anytime the return value handlers side effect on the parent's state. Parallel tasks remove the non-determinism with side-effecting return value handlers that make data-race detection especially hard while preserving the core semantics that encompass most aspects of existing programming models. For example, languages, such as OpenMP, Cilk, X10, Habanero, fall largely within the semantics of parallel tasks. The only real notable missing language features are barriers for phased execution and some generality in futures. Barriers for phased execution are a lesser known and used language feature unique to Habanero. The well-known and widely used core language features are covered by parallel tasks.

4.1 Syntax

The syntax for parallel tasks is given in Fig. 3. A program **P** is a sequence of procedures with names taken from a finite set: $p_0 \ldots p_i \in \mathtt{Procs}^*$. Each p has a single L-type parameter `l` taken from a finite set of parameter names `Vars` and a top-level statement **s** that gives the body of the procedure. The semantics is abstracted over concrete values and operations, so the possible types of `l` are not specified.

Statements are inductive via composition with other control-flow statements, and the set of all statements is given by `Stmts`. The syntax for expressions is intentionally undefined but includes a finite set of values `Vals` and local or global variable references. It is assumed that `Vals` minimally includes **true**, **false**, and a special value $\perp$ for uninitialized. Global variable references are taken from a finite set of names, `Globals`, where $\mathtt{Globals} \cap \mathtt{Vars} = \emptyset$. The names include a special reserved variable `isolate` that is only used by the semantics for mutual exclusion.

The **async** , **future** , **await** , and **isolated** statements relate to concurrency; the rest of the statements are sequential and have their usual meaning. Intuitively, the **async** statement adds a task into a region r, taken from a finite set of region identifiers, `Regs`, by indicating the procedure p for the task with an expression e for the value of the local variable. The return value from p is ignored and not combined into the parent state. The **future** statement is similar only it includes a return value handler $d : \mathtt{Vals} \rightarrow \mathtt{Stmts}$ to combine a return value into the parent state, and it allows the parent to pass the ownership of tasks in the regions in the region sequence **r** to the child.

A task is termed *completed* when its has evaluated the expression contained in its **return** statement. The **await** statement blocks execution until some task with a handle in r completes at which point its return handler is executed. The **await** statement recursively calls itself until all tasks with handles in r complete.

The **isolated** statement provides mutual exclusion relative to other **isolated** statements. If s is isolated, then it runs mutually exclusive to any other statements s' that are also isolated; however, s does not run mutually exclusive to other non-isolated statements that may be concurrent with s. An **isolated** statement must consist of only sequential statements.

For the rest of the presentation, only well-formed programs are considered.

Definition 2 *(Well-formed)* A program is said to be *well-formed* if and only if it conforms to a programming model that ensures (1) freedom from deadlock; and (2) awaiting futures cannot introduce non-determinism.

There are several programming models guarantee well-formedness. For example, consider the Habanero model. For deadlock, there is only a single lock for mutual exclusion, so it is not possible to *hold and wait*, and it prohibits cyclic dependencies between futures and only allows sequential statements in isolation for the same reason. Additionally, futures do not have return value handlers; instead, the value returned from a future's statement block is made accessible via the future's `get()` method. As a result, no nondeterminism is introduced when a **finish** block waits on multiple futures.

The language is purposely kept simple to focus on the core concept of a computation graph as an equivalence class over task-parallel program executions. Additionally, there is no loss of generality in restricting procedures to a single parameter or not including statements for procedure calls because, for example, there are global variables, the syntax for types and expressions is left free, and procedures calls are syntactic sugar for a **future** statement followed by an **await** statement with an appropriate return-value handler.

4.2 Tree-based semantics

The semantics of task-parallel programs is defined over a tree of procedure frames to represent the concurrency in the language rather than a stack of procedure frames which is inherently sequential. A frame in the tree is a *task*. Execution proceeds by stepping some task in the tree. If that step creates a new task, then the task is posted as a child of the creating task in the tree. If that step consumes a completed child task, then the completed child task is removed from the tree, and any unconsumed children of the consumed task become children of the consuming task.

Additional to the tree of tasks is a global store and a computation graph. The computation graph is additional to the program state, and that state is updated by the semantics to capture the program execution; it includes a few additional tuple members that will be defined shortly. The computation graph has no bearing on how program execution proceeds, that is completely defined by the tree of tasks and the global store.

Any task in the tree that is not blocked may step in an execution of a task-parallel program. A task step updates the auxiliary computation graph as appropriate, and it updates the statement for the current task. Additionally, depending on the statement, the step may also change the value of the task's local variable, change the global store, or, as mentioned previously, change the structure of the tree by adding or removing a child. With this intuition, the semantics are ready to be formalized.

4.2.1 Program state

The *state* of a parallel tasks program is defined by the tuple $(\Gamma, \sigma, (t, m))$ where Γ is an augmented computation graph, $\sigma : \texttt{Globals} \to \texttt{Vals}$ is a partial function mapping global variable names to values, and (t, m) is a tree configuration for the root of the tree. Let σ_o be the initial store such that $\forall g \in \texttt{Globals}, \sigma_o(g) = \bot$.

An *augmented computation graph* is a tuple $\Gamma = (G, last, R)$, where $G = (N, E, \rho, \omega)$ is a computation graph; $last \in N$ is the last isolated node and is used to assert the observed order of **isolated** statements in the execution; and $R : \texttt{Regs} \mapsto \mathcal{P}(N)$ is a function to track nodes that need to join in the computation graph at the end of an **await** statement. In general, a function notation is adopted to access members of tuples. For example, the members of G are accessed as $G(N)$, $G(E)$, $G(\rho)$, etc. Let Nodes be a finite set of nodes, and fresh() return as yet unused nodes in Nodes. Let Γ_o be the initial augmented computation graph such that for $n_o = \text{fresh}()$, $\Gamma_o(G) = (\{n_o\}, \emptyset, \rho_o, \omega_o)$ where $\forall n \in \texttt{Nodes}, \rho_o(n) = \emptyset \wedge \omega_o(n) = \emptyset$, $\Gamma_o(last) = n_o$, and $\forall r \in \texttt{Regs}, \Gamma_o(R)(r) = \emptyset$.

A *tree configuration*, $c = (t, m)$, is an inductively defined tree with task-labeled vertices, t, and region labeled edges given by the *region valuation* function, $m : \texttt{Regs} \to \texttt{Configs}$, where Configs is the set of tree configurations. The tree is finite and unordered. For a given vertex $c = (t, m)$, $m(r)$ returns the collection of sub-trees connected to the t-labeled root by r-labeled edges. In this context, m is local to the current task. Let m_o be the initial region valuation such that $\forall r \in \texttt{Regs}, m_o(r) = \emptyset$; it is used in both the initial state and transition rules when creating new tree configurations.

A task $t = (\ell, s, d, n)$ is a tuple containing the value, $\ell \in \texttt{Vals}$, of the task's single local variable l, along with its statement s, its return value handler d, and its associated node n in the computation graph. The initial task $t_o = (\bot, s_o, \lambda v.\textbf{skip}, \Gamma_o(last))$ is defined to start and then await the initial task,

$$s_o = \textbf{async}\ \ r_o \leftarrow p_o\ \ell\ ;\ \ \textbf{await}\ \ r_o$$

where p_o is the top level procedure and ℓ is the initial value for p_o's single parameter. With that, the initial state of a task-parallel program is $(\Gamma_o, \sigma_o, (t_o, m_o))$.

4.3 Notation

The semantics are defined as a set of transition rules relating states. Some amount of additional notation is needed to help the presentation of the transition rules be as concise and uncluttered as possible. That notation follows.

Let $[\![\cdot]\!]_e$ be a partial evaluation function for expressions without any variables. For convenience in the semantics:

$$\begin{aligned} e(t, \sigma) &= e((\ell, s, d, n), \sigma) \\ &= e(\ell, \sigma) \\ &= [\![e[\ell/\texttt{l}, \sigma(\texttt{g}_0)/\texttt{g}_0, \sigma(\texttt{g}_1)/\texttt{g}_1, \ldots]]\!]_e \end{aligned}$$

If $e[\ell/\texttt{l}, \sigma(\texttt{g}_0)/\texttt{g}_0, \sigma(\texttt{g}_1)/\texttt{g}_1, \ldots]$ has any free variables or other errors, then by definition, $[\![e[\ell/\texttt{l}, \sigma(\texttt{g}_0)/\texttt{g}_0, \sigma(\texttt{g}_1)/\texttt{g}_1, \ldots]]\!]_e$ has no meaning and is undefined. The set of global variables that appear in an expression is given by $\texttt{Globals}(e)$.

A *statement context* S for s is a convenient way to consider, and manipulate, the next-to-be-executed statement in s. For example, $S[\texttt{l} := e]$ means that the next-to-be-executed statement in the context S is $\texttt{l} := e$; other statements may follow since s is inductive (e.g., $\texttt{l} := e\ ;\ s'$). A *configuration context*, C, is a way to consider a single task in a larger tree leaving the surrounding context (i.e., other tasks in the tree) unchanged.

The contexts are used to express transition rules. For example, consider the transition rule for a step on task whose next-to-execute statement is a **skip**. The transition rule, via contexts, is given as:

$$\frac{}{\begin{array}{c}(\Gamma, \sigma, C[(\ell, S[\textbf{skip}\ ; s], d, n), m]) \\ \to (\Gamma, \sigma, C[(\ell, S[s], d, n)\, m])\end{array}}\ \textsc{Skip}$$

The configuration context can be any task in the tree that matches the statement context. In the source state, it is matched when the next-to-execute instruction in the statement context is **skip**; s where s is any valid statement. In the target state, the transition side effects the task's statement context to be s, having consumed the **skip**, and leaves the rest of the surrounding context, or state, unchanged.

A function can be *strongly updated* as in a write to a global store: $\sigma' = \sigma[\mathtt{g} \mapsto \ell]$; the new store is just like the old store only now $\sigma'(\mathtt{g}) = \ell$. A function may also be *weakly updated* as in adding new variables into the read set for a node in the computation graph: $\rho' = \rho[n \overset{\cup}{\mapsto} \mathtt{Globals}(e)]$; the new read set is just like the old read set only now $\rho'(n) = \rho(n) \cup \mathtt{Globals}(e)$.

A function may remove mappings as in a parent task consuming a completed child task in the region valuation: $m'_1 = m \backslash (r \mapsto (t_2, m_2))$; the new region valuation is just like m only now $m'_1(r)$ no longer includes (t_2, m_2). Regions may also be merged together as in $m_1 \cup m_2$ with the intuitive weak update meaning. The projection of the region valuation, m, to the sequence of regions $\mathbf{r}$ is given as $m|_{\mathbf{r}}$ and defined such that $m|_{\mathbf{r}}(r') = m(r')$ when r' occurs in $\mathbf{r}$ and $m|_{\mathbf{r}}(r') = \emptyset$ otherwise.

And finally, for a tuple such as a $\Gamma = (G, \mathit{last}, R)$, this presentation adopts the same notation for updating functions where $\Gamma' = \Gamma[\mathit{last} \mapsto n']$ is understood to mean that $\Gamma'(\mathit{last})$ now has the value n' and everything else is the same as in Γ.

There are two additional functions to help manipulate the augmented computation graph efficiently in the transition rules. The *fork* function adds nodes for new tasks: $G' = \mathrm{fork}(G, n, n', n'', V)$ forks the node n in the computation graph G to n' and n'' and updates the read seat for n with the variables in the set V:

$$\begin{aligned} G' = G[N &\overset{\cup}{\mapsto} \{n', n''\}] \\ &[E \overset{\cup}{\mapsto} \{(n, n'), (n, n'')] \\ &[\rho(n) \overset{\cup}{\mapsto} V] \end{aligned}$$

The new graph has a new thread of concurrent execution and corresponds to the creation of a new task in the task configuration tree.

The *join* function makes clear in the graph where completed tasks synchronize at an **await** statement: $G' = \mathrm{join}(G, R, n_2, n)$ joins the nodes in R and the node from the recently completed task n_2 to the new node n in the computation graph G to n':

$$\begin{aligned} G' = G[N &\overset{\cup}{\mapsto} \{n\}] \\ &[E \overset{\cup}{\mapsto} \{(n, n_2)\} \cup \{(n, n_i) \mid n_i \in R\}] \end{aligned}$$

The new graph has gathered concurrent executions at the join and corresponds to the removed child tasks in the configuration tree.

4.3.1 Transition rules

Figure 4 gives the transition rules for all the *sequential* statements but the **skip** statement (given previously). Sequential statements have no direct affect on the shape of the configuration tree. Relative to the computation graph, these statements are only able to update read or write sets for the task's associated computation graph node.

The **return** statement does merit some little discussion. Its rule inserts a machine only **done** statement. The rule evaluates the return expression and then puts the value in the **done** statement. Later, when an **await** statement joins with the completed task, the **done** statement is consumed and the value is used by the return-value handler.

$$\textsc{Assign Local}\quad \frac{\mathtt{l} \notin \mathtt{Globals} \qquad \ell' = e(\ell, \sigma) \qquad \Gamma' = \Gamma[G(\rho)(n) \overset{\cup}{\mapsto} \mathtt{Globals}(e)]}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathtt{l} := e], d, n), m]) \rightarrow \\ (\Gamma', \sigma, C[(\ell', S[\mathbf{skip}], d, n), m])\end{array}}$$

$$\textsc{Assign Global}\quad \frac{\mathtt{l} \in \mathtt{Globals} \qquad \sigma' = \sigma[\mathtt{l} \mapsto e(\ell, \sigma)] \qquad \Gamma' = \Gamma[G(\rho)(n) \overset{\cup}{\mapsto} \mathtt{Globals}(e)][G(\omega)(n) \overset{\cup}{\mapsto} \{\mathtt{l}\}]}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathtt{l} := e], d, n), m]) \rightarrow \\ (\Gamma', \sigma', C[(\ell, S[\mathbf{skip}], d, n), m])\end{array}}$$

$$\textsc{If-then}\quad \frac{\mathbf{true} = e(\ell, \sigma) \qquad \Gamma' = \Gamma[G(\rho)(n) \overset{\cup}{\mapsto} \mathtt{Globals}(e)]}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2], d, n), m]) \rightarrow \\ (\Gamma', \sigma, C[(\ell, S[s_1], d, n), m])\end{array}}$$

$$\textsc{If-else}\quad \frac{\mathbf{false} = e(\ell, \sigma) \qquad \Gamma' = \Gamma[G(\rho)(n) \overset{\cup}{\mapsto} \mathtt{Globals}(e)]}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2], d, n), m]) \rightarrow \\ (\Gamma', \sigma, C[(\ell, S[s_2], d, n), m])\end{array}}$$

$$\textsc{Do-loop}\quad \frac{\mathbf{true} = e(\ell, \sigma) \qquad \Gamma' = \Gamma[G(\rho)(n) \overset{\cup}{\mapsto} \mathtt{Globals}(e)]}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathbf{while}\ e\ \mathbf{do}\ s], d, n), m]) \rightarrow \\ (\Gamma', \sigma, C[(\ell, S[s;\ \mathbf{while}\ e\ \mathbf{do}\ s], d, n), m])\end{array}}$$

$$\textsc{Do-break}\quad \frac{\mathbf{false} = e(\ell, \sigma) \qquad \Gamma' = \Gamma[G(\rho)(n) \overset{\cup}{\mapsto} \mathtt{Globals}(e)]}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathbf{while}\ e\ \mathbf{do}\ s], d, n), m]) \rightarrow \\ (\Gamma', \sigma, C[(\ell, S[\mathbf{skip}], d, n), m])\end{array}}$$

$$\textsc{Return}\quad \frac{v = e(\ell, \sigma) \qquad \Gamma' = \Gamma[G(\rho)(n) \overset{\cup}{\mapsto} \mathtt{Globals}(e)]}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathbf{return}\ e], d, n), m]) \rightarrow \\ (\Gamma', \sigma, C[(\ell, S[\mathbf{done}\ v], d, n), m])\end{array}}$$

Fig. 4 Transition rules for sequential statements

$$\frac{\begin{array}{l}\text{ASYNC}\\ \ell = e(\ell', \sigma) \qquad d = \lambda v.\mathbf{skip} \qquad n_0', n_1 = \text{fresh}()\\ m' = m[r \overset{\cup}{\mapsto} ((\ell, s_p, d, n_1), m_o)]\\ \Gamma' = \Gamma[G \mapsto \text{fork}(\Gamma(G), n_0, n_0', n_1, \texttt{Globals}(e))]\end{array}}{\begin{array}{l}(\Gamma, \sigma, C[(\ell', S[\mathbf{async}\ r \leftarrow p\ e], d', n_0), m]) \rightarrow\\ (\Gamma', \sigma, C[(\ell', S[\mathbf{skip}], d', n_0'), m'])\end{array}}$$

$$\frac{\begin{array}{l}\text{FUTURE}\\ \ell = e(\ell', \sigma) \qquad n_0', n_1 = \text{fresh}()\\ m' = (m \setminus m|_{\mathbf{r}})[r \mapsto ((\ell, s_p, d, n_1), m|_{\mathbf{r}})]\\ \Gamma' = \Gamma[G \mapsto \text{fork}(\Gamma(G), n_0, n_0', n_1, \texttt{Globals}(e))]\end{array}}{\begin{array}{l}(\Gamma, \sigma, C[(\ell', S[\mathbf{future}\ r \leftarrow p\ e\ d], d', n_0), m]) \rightarrow\\ (\Gamma', \sigma, C[(\ell', S[\mathbf{skip}], d', n_0'), m'])\end{array}}$$

$$\frac{\begin{array}{l}\text{AWAIT}\\ m_1 = m \setminus (r \mapsto ((\ell_2, S[\mathbf{done}\ v], d_2, n_2), m_2))\\ s = d(v) \qquad (m_1 \cup m_2)(r) \neq \emptyset\\ \Gamma' = \Gamma[R(r) \overset{\cup}{\mapsto} \{n_2\}]\end{array}}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathbf{await}\ r], d, n), m]) \rightarrow\\ (\Gamma', \sigma, C[(\ell, S[s;\ \mathbf{await}\ r], d, n), m_1 \cup m_2])\end{array}}$$

$$\frac{\begin{array}{l}\text{AWAIT-DONE}\\ m_1 = m \setminus (r \mapsto ((\ell_2, S[\mathbf{done}\ v], d_2, n_2), m_2))\\ s = d(v) \qquad (m_1 \cup m_2)(r) = \emptyset \qquad n' = \text{fresh}()\\ \Gamma' = \Gamma[R(r) \mapsto \emptyset][G \mapsto \text{join}(\Gamma(G), \Gamma(R), n_2, n')]\end{array}}{\begin{array}{l}(\Gamma, \sigma, C[(\ell, S[\mathbf{await}\ r], d, n), m]) \rightarrow\\ (\Gamma', \sigma, C[(\ell, S[s], d, n'), m_1 \cup m_2])\end{array}}$$

$$\frac{\begin{array}{l}\text{ISOLATED}\\ \sigma(\texttt{isolate}) = \mathbf{false} \qquad \sigma' = \sigma[\texttt{isolate} \mapsto \mathbf{true}]\\ n' = \text{fresh}()\\ \Gamma' = \Gamma[G(N) \overset{\cup}{\mapsto} \{n'\}][G(E) \overset{\cup}{\mapsto} \{(n, n'), (\Gamma(last), n')\}]\end{array}}{\begin{array}{l}(\Gamma, \sigma, C[(\ell', S[\mathbf{isolated}\ s], d', n), m]) \rightarrow\\ (\Gamma', \sigma', C[(\ell', S[s; \mathbf{isolated\text{-}end}], d', n'), m])\end{array}}$$

$$\frac{\begin{array}{l}\text{ISOLATED-DONE}\\ \sigma(\texttt{isolate}) = \mathbf{true} \qquad \sigma' = \sigma[\texttt{isolate} \mapsto \mathbf{false}]\\ n' = \text{fresh}()\\ \Gamma' = \Gamma[G(N) \overset{\cup}{\mapsto} \{n'\}][G(E) \overset{\cup}{\mapsto} \{(n, n')\}][last \mapsto n]\end{array}}{\begin{array}{l}(\Gamma, \sigma, C[(\ell', S[\mathbf{isolated\text{-}end}], d', n), m]) \rightarrow\\ (\Gamma', \sigma', C[(\ell', S[\mathbf{skip}], d', n'), m])\end{array}}$$

Fig. 5 Transition rules concurrent statements

Figure 5 gives the transition rules for all the concurrent statements in the language. These statements affect the shape of the configuration tree and the shape of the computation graph.

The ASYNC rule creates a new child task. The rule adds two fresh nodes n_0' and n_1 to the computation graph. Node n_0' represents the statements following the **async** and n_1 represents the statements to be executed by the new task. The rule orders both after the current node for the parent, n_0, in the computation graph. The read set ρ of node n_0 is updated to include any global variables referenced in the expression, `Globals` (e), for the local parameter value in the new task. The region mapping m of the parent task is updated to include the new task with its empty initial region mapping. The FUTURE rule mimics the ASYNC rule only it allows arbitrary return value handlers, and its region valuation is seeded with regions passed from the parent using projection.

The AWAIT rule blocks the execution of the currently executing task until a task in the indicated region completes. A new node to join the two tasks is not created in the computation graph, nor are the two tasks ordered in the sense of join because the choice of task t_2 in the region is nondeterministic; as such, the computation graph allows tasks in the region to join in any order contrary to the observed reduction by the rule. The rule saves the current node in the graph for t_2, n_2, to join later once the region is empty, and it updates the read set for t_2 on the expression in the **return** statement.

The rest of the rule separates out the task from the region, makes sure the region is not yet empty, and gets the statement for the return value handler. The new state adds an **await** statement after the return value handler statement since the region is not yet empty, and the region valuation function in the new state includes any tasks owned by t_2.

The AWAIT- DONE activates when the last task in the region is joined. It differs from the AWAIT rule in that it orders all tasks that have joined in the region to happen-before the new node for the parent in the computation graph, and it does not insert another **await** statement in the new state.

If no other isolated statements are running, then the ISOLATED rule updates the `isolated` shared variable and inserts after the isolated statement s the new **isolated-end** keyword that is only a part of the machine syntax for the semantics. The computation graph gets a new node to track accesses in the isolated statement with an appropriate edge from the previous node. A sequencing edge from *last* is also added so the previous isolated statement happens before this new isolated statement. As a reminder, *last* is initialized to the initial node when execution starts.

The ISOLATED- END rule creates a new node in the computation graph to denote the end of isolation, updates the `isolated` shared variable, and it updates *last* to properly sequence any future isolation. The *last* variable implements how the semantics track and record the order of **isolated** statements in the computation graph while reducing the state.

5 Proof of correctness

The semantics in Sect. 4 determine both how a program executes and how computation graphs are created. This section proves that computation graphs are correct with respect to the executions that produce them. It also proves a much stronger claim: Computation graphs are correct with respect to all compatible executions. A computation graph contains

a data-race if and only if an execution with the same order on **isolated** statements contains a data-race.

This section begins with relevant formal definitions, including a definition for data-race. It then proceeds to relevant reasoning, concluding with Theorem 4, which proves soundness, and Theorem 5, which proves completeness.

Definition 3 *(Compatible execution)* A given execution is compatible with a computation graph G if and only if its order on **isolated** statements is the same as the order on isolated nodes in G.

Definition 4 *(Conflict)* Two statements conflict if they both access the same shared variable and at least one of them writes to that variable. $\rho(s)$ and $\omega(s)$ behave as expected.

$$\text{conflict}(s_i, s_j) = \begin{array}{l} \rho(s_i) \cap \omega(s_j) \neq \emptyset \vee \\ \rho(s_j) \cap \omega(s_i) \neq \emptyset \vee \\ \omega(s_i) \cap \omega(s_j) \neq \emptyset. \end{array} \tag{2}$$

A state's set of active statements is used to define concurrency.

Definition 5 *(Active statements)* A state ς has a set of active statements $a(\varsigma)$ that corresponds to the next statement to be reduced in each of the active tasks in the state.

Notice that the definition does not check if the active statements are reducible. For example, an **await** statement may be active but not reducible in a state because there is no task in its indicated region that has completed; thus, the AWAIT rule is not active on that statement. This nuance becomes important in the next definition. For that definition, and without loss of generality, we call the program's initial state ς_0.

Definition 6 *(Concurrency)* Any two statements are concurrent if and only if an execution of the program can result in a state ς such that both statements are active at the same time:

$$s \mid\mid s' \iff \exists \varsigma : \varsigma_0 \rightarrow^* \varsigma \ \wedge \ \{s, s'\} \subseteq a(\varsigma). \tag{3}$$

A state ς that satisfies this condition for s and s' is called a witness state for $s \mid\mid s'$. $\rightarrow^*$ is the transitive closure of the transition relation $\rightarrow$ defined in Sect. 4.2. Two statements are ordered if and only if they are not concurrent.

Note that the definition treats some statements as concurrent that are in fact not concurrent in the witness state. For example, consider an **isolated** statement and an **assign** statement. The **isolated** statement may be blocked in the witness state waiting for the isolation lock while the **assign** statement is always able to reduce; hence, these statements are not actually concurrent in the witness state.

This discrepancy is harmless because the only statements that can block in these ways are the **isolated** statement and the **await** statement. In the case of the **isolated** statement, it only interacts with the machine only **isolated** global variable. Similarly, a task that is completed is at a **done** statement, and that statement does not read or write any variables; reading and writing variables happens when the **return** statement reduced. As such, it is not possible to ever have a data-race with either of these statements, so calling them concurrent with other statements is harmless.

Definition 7 *(Data race)* There is a data-race on two statements s and s' if and only if they conflict and are concurrent:

$$DR(s, s') = s \mid\mid s' \ \wedge \text{conflict}(s, s'). \tag{4}$$

Two statements that occur in the same thread of execution cannot be concurrent, as exactly one statement is active in each active thread at any point in time. Accordingly, events contained in the same node are always serialized. The semantics for the ISOLATED and ISOLATED- DONE transitions in Fig. 5 ensure that no two statements inside of **isolated** statements can be concurrent, as only one thread may enter an **isolated** statement at a time.

Definition 8 *(Specific transition)* A state ς transitions to some other state ς' via a configuration context C when C is the configuration context reduced in the transition from ς to ς'. This is denoted $\varsigma \xrightarrow{C} \varsigma'$.

Definition 9 *(Reachable states)* A set Σ_r of reachable states is the set of states reachable from the initial state ς_0:

$$\Sigma_r \equiv \{\varsigma \mid \varsigma_0 \rightarrow^* \varsigma\}. \tag{5}$$

Definition 10 *(Safe statement ordering)* Any two statements s and s' are safely ordered if and only if any state ς with two configuration contexts C and C' transitions to the same state ς' after transitioning via C and C', regardless of the order in which these transitions occur:

$$\begin{aligned} s \overset{?}{\leftrightarrow} s' \equiv \forall \varsigma \in \Sigma_r : (\varsigma(C) = ((l, s, d, n), m) \wedge \\ \varsigma(C') = ((l', s', d', n'), m')) \\ \implies \varsigma \xrightarrow{C} \varsigma_1 \xrightarrow{C'} \varsigma' \wedge \varsigma \xrightarrow{C'} \varsigma_2 \xrightarrow{C} \varsigma'. \end{aligned} \tag{6}$$

Two statements can be safely ordered in one of two ways. First, they can be ordered (in which the definition is true vacuously). Second, they can commute, satisfying the definition's consequent. Thus, all realizable orders on safely ordered statements yield deterministic results.

A data-race may introduce nondeterminism. Similarly, it is possible for conflicting statements (intended data-races) in different **isolated** statements to introduce nondeterminism. Lemma 1 and Theorem 3 prove that in the absence of both data-race and isolation, program execution is deterministic.

Lemma 1 *(Local determinism) When two statements do not race, they are safely ordered:*

$$\neg DR\left(s, s'\right) \implies s \overset{?}{\leftrightarrow} s'. \tag{7}$$

Proof A well-formed program cannot introduce data-race on the order in which futures are gathered. In a well-formed program, then, this assertion is equivalent to two implications:

$$\begin{aligned}
&\neg DR\left(s, s'\right) \implies s \overset{?}{\leftrightarrow} s' \\
&\neg\left(s \mid\mid s' \wedge \text{conflict}\left(s, s'\right)\right) \implies s \overset{?}{\leftrightarrow} s' \\
&\left(s \mid\mid s' \wedge \text{conflict}\left(s, s'\right)\right) \vee s \overset{?}{\leftrightarrow} s' \\
&\left(s \mid\mid s' \vee s \overset{?}{\leftrightarrow} s'\right) \wedge \left(\text{conflict}\left(s, s'\right) \vee s \overset{?}{\leftrightarrow} s'\right) \\
&\left(\neg s \mid\mid s' \implies s \overset{?}{\leftrightarrow} s'\right) \wedge \left(\neg\text{conflict}\left(s, s'\right) \implies s \overset{?}{\leftrightarrow} s'\right)
\end{aligned}$$

The first implication is true from Definition 10, when s and s' are safely ordered because no state exists in which both are active.

The second implication is true by induction on the transition rules in Figs. 4 and 5. Informally, the two statements act independently in their respective configuration contexts. Only a global side effect in one statement that changes a value read or written in the other statement could change the second statement's behavior. This can only occur, however, when the two statements conflict.

For example, consider a state with two active statements, one whose next transition is governed by the ASSIGN LOCAL transition rule and another whose transition is governed by the ASSIGN GLOBAL rule. The first statement computes the value of e and the second changes the value of `l`. If the value of e depends on `l`, the two statements conflict and the implication is vacuously true. On the other hand, if the value of e does not depend on that of `l`, its result is the same regardless of whether the ASSIGN GLOBAL transition happens before or after the ASSIGN LOCAL transition. □

Theorem 3 (General determinism) *In the absence of data-race and on some order of* ***isolated*** *statements, a program's behavior is deterministic.*

Proof By induction on Lemma 1. □

Definition 11 *(Race-sensitive partial order)* Every computation graph is derived from some execution, which is a total order on statements executed. Some of these statements race with each other; by definition, these statements occur in pairs. The computation graph is a partial order on the statements contained in its nodes. This partial order, together with additional pairs to enforce the observed order on pairs of statements that race with each other (and additional pairs, as needed, for transitivity) is a race-sensitive partial order.

Race-sensitive partial orders are not created in practice. However, they are useful as a theoretical construct for the proof. Observe that whenever there is no data-race, there are no additional synchronization pairs and the race-sensitive partial order is the same as the order induced by the computation graph.

Lemma 2 *If two nodes n and n' are ordered in a computation graph G, every $s \in n$ and $s' \in n'$ are ordered:*

$$n \prec_G n' \implies s \prec s'. \tag{8}$$

Proof By induction on the transition rules in Fig. 5 (no rule in Fig. 4 creates a node or changes its task's node except to add a statement to that node). Every computation graph G is created by these transition rules. Every edge is created from a node that has completed (no additional events are added to it) to a node that is beginning, with the exception of *last*, which is used only to create an edge in the ISOLATED transition rule, no state contains a reference to n after the ASYNC, FUTURE, AWAIT- DONE, ISOLATED, or ISOLATED-DONE rules complete. In each of these cases, the edge or edges created in the transition reflect a necessary order on events. For example, an **async** statement (in n_0) must execute before the events that follow it in its own task (in n'_0) and must occur before all events in the task it creates (in n_1). Events contained in the same node always pertain to the same task and are strictly ordered.

Accordingly, every edge in every computation graph reflects an ordering imposed by the semantics. □

Corollary 1 *If two statements are unordered, their respective nodes in the computation graph are unordered:*

$$s \mid\mid s' \implies n \mid\mid_{\prec_G} n' \tag{9}$$

Proof By two uses of the contrapositive of Lemma 2:

$$\begin{aligned}
&s \not\prec s' \implies n \not\prec_G n' \wedge s' \not\prec s \implies n' \not\prec_G n \\
&s \not\prec s' \wedge s' \not\prec s \implies n \not\prec_G n' \wedge n' \not\prec_G n \\
&s \mid\mid s' \implies n \mid\mid_{\prec_G} n'.
\end{aligned}$$

□

Lemma 3 reasons about unordered nodes in a race-sensitive partial order. This is in contrast with Lemma 2, which reasons about ordered nodes in a computation graph.

Lemma 3 *If two nodes n and n' are unordered in a computation graph G, every $s \in n$ and $s' \in n'$ not ordered in the race-sensitive partial order induced by G are concurrent:*

$$\forall s \in n, s' \in n' : n \mid\mid_{\prec} n' \implies s \mid\mid s'. \tag{10}$$

Proof If s and s' are ordered in the race-sensitive partial order for G, the lemma holds vacuously.

Synchronization between threads is accomplished by restrictions in the transition rules. For example, no task may enter an **isolated** statement when another task is currently inside an **isolated** statement of its own because the other task must have set `isolate` to **true** when executing the ISOLATED transition rule and no other ISOLATED transition rule may fire before the task currently in a critical section exits, setting `isolate` to **false** in its execution of the ISOLATED-DONE rule. Similarly, an **await** statement cannot step forward (per the AWAIT and AWAIT- DONE transition rules) until one of the tasks in its region is at a **done** statement (and is therefore finished) and cannot complete until all tasks in its region have been reaped (per the AWAIT- DONE transition rule). In every case, execution of a synchronization event results in the creation of a corresponding edge in G.

There is a single first node in each computation graph G that happens before every other node. As a result, G is a finite semilattice with a global infimum. It follows that every two nodes n and n' have a least common ancestor n_a.

Since s and s' both exist in G, they are both reachable in the execution that created G. The same is true of n and n'. As a result, their least common ancestor n_a is also reachable. Therefore, some state ς_a must exist that creates outgoing edges to n_1 and n_2 where n_1 is either equal to n or is an ancestor of it and where the same is true of n_2 and n'. N_1 is the set of nodes on a path from n_1 to n, inclusive. N_2 is the set of nodes on a path from n_2 to n', inclusive. There cannot be any synchronization edges between nodes in N_1 and N_2, as this would contradict the definition of n_a as the least common ancestor of n and n'.

As a result, it is possible for ς_a to transition in the two tasks independently of each other until reaching s and s'. The state in which both statements are active is a witness for their concurrency. □

Corollary 2 *Every topological sort on a race-sensitive partial order is a realizable trace.*

Proof By Lemma 3. □

Theorem 4 (Soundness) *If a computation graph G contains a data-race, an actual data-race exists on an execution compatible with that graph.*

Proof The race-sensitive partial order for G is $\prec_r$. Let S_r be the set of conflicting statements in unordered nodes in G; that is, the statements that G claims to be racing statements. Because G contains a data-race, S_r must have at least two members. Furthermore, Corollary 1 proves that every pair of statements that actually race must belong, respectively, to unordered nodes. Since they conflict by definition, these statements are members of S_r.

Let a reported race be a pair of members of S_r that conflict with each other and that occur in nodes that are unordered in G. Let s and s' be a reported race such that s occurs before s' in the observed order of execution. Let n and n', respectively, be the nodes that contain s and s'. Without loss of generality, choose s' to be the first statement that meets these requirements. Again without loss of generality, choose s to be the last statement compatible with these requirements (including the choice of s'). To restate, *conflict* (s, s') and $n \mid\mid_{\prec_G} n'$.

By Corollary 2, any topological sort on $\prec_r$ is a realizable execution. Choose a topological sort that executes until s is active and then advances other tasks until s' is active. The choice of s and s' ensures that if they are ordered by $\prec_r$, they race: if they were ordered because of some other pair of statements that races, the second of those statements must precede s', creating a contradiction. By Lemma 3 and because n and n' are unordered, s and s' must either race each other (in which case the lemma holds) or must be unordered by $\prec_r$. This latter case, however, contradicts itself: if s and s' are unordered in $\prec_r$, they are necessarily concurrent. Since it is given that they conflict, they race. This, however, contradicts that s and s' are unordered by $\prec_r$, which orders all racing statements. □

Theorem 5 (Completeness) *If a computation graph G does not contain a data-race, no compatible execution contains a data-race.*

Proof By Lemma 2, any statement in two ordered nodes are ordered. If statements in unordered nodes conflict, there is a data-race in the graph. Therefore, if there is no data-race in a graph, no statements in its unordered nodes conflict and no data-race is possible. □

6 Model checking task-parallel programs

This section applies model checking to a task-parallel program to prove data-race freedom under the given input to the program. The model checker uses a depth-first search-based algorithm to exhaustively explore all computation graphs that arise from different total orders on isolated nodes. A proof of correctness is given. The proof shows that the depth-first search enumerates a sufficient set of graphs to be both sound and complete on the given input.

6.1 Depth-first scheduling

The depth-first algorithm to enumerate all schedules over **isolated** statements is given in Algorithm 2. The algorithm is a fairly direct implementation of a depth-first search on the semantics. It is divided into five ordered cases:

1. The program is completed and only a single task is left that cannot reduce any further (line 2);

2. There is some task that is not at an **await** statement or **isolated** statement that can be reduced (possibly more than once);
3. There is some task at an **await** statement that can be reduced;
4. There is some task at an **isolated** statement that can be reduced from which all possible choices must be explored; or
5. The program is not well-formed.

Reduced in this context means the state has a successor. Each case is discussed below.

line 2 detects when the task-parallel program has run to completion. The function $\texttt{Configs}(\varsigma)$ returns all configuration contexts (i.e., task frames) in the tree in the state, and the notation $\varsigma \nrightarrow \varsigma'$ is intended to convey that it is not possible to reduce the state any further. A well-formed program gathers all tasks eventually, which means eventually there is as single context in the task tree, and the statement in that context is a **skip** statement that cannot be reduced because there is no statement that follows. In this case, the computation graph is pulled out of the state and checked for data-race.

Line 4 reduces a context in the tree that is not at an **await** statement or **isolated** statement. The notation $\varsigma \overset{C}{\hookrightarrow} \varsigma'$ is intended to convey that the context is reduced as much as possible, which is at least one time, using any of the rules in Fig. 4 or Fig. 5 but AWAIT, AWAIT- DONE, and ISOLATED; the state is maximally reduced when done with the reductions and should be stopped at the following: a **done** statement, **await** statement, **isolated** statement, or the case in line 2. In all of these cases, the procedure makes a recursive call on the reduced state.

line 6 reduces **await** statements. It is only reached when the first two cases fail; meaning the program is not able to reduce further without first reducing either an **await** statement or **isolated** statement. The algorithm first gives priority to the **await** statement. It this case, if it is possible to step the context exactly once on the **await** statement, then the procedure makes the recursive call on the successor state.

line 8 is the depth-first search over the **isolated** statements. The key difference with regard to the await case on line 6 is that this case searches from the successor of every context with an **isolated** statement that can reduce at the current state; not just one successor. As such, the current state is the backtrack point for the search; thus, from this state, all possible orders of isolation are considered.

The final case on line 12 should only be reached in the case of a program that is not well formed. Such a program may be deadlocked, have dangling tasks that have not been gathered, etc. Note that a divergent program will cause an infinite loop and not be detected.

Algorithm 2 The algorithm to enumerate all schedules over isolation.

```
1: procedure DFS(ς)
2:   if |Configs(ς)| == 1 ∧ ς ↛ ς′ then
3:     DetectRace(ς(Γ)(G))
4:   else if ∃C ∈ Configs(ς) (¬await(C) ∧ ¬isolated(C) ∧ ς ↪C ς′) then
5:     DFS(ς′)
6:   else if ∃C ∈ Configs(ς) (await(C) ∧ ς →C ς′ then
7:     DFS(ς′)
8:   else if ∃C ∈ Configs(ς) (isolated(C) ∧ ς →C ς′) then
9:     for all C ∈ Configs(ς) (isolated(C) ∧ ς →C ς′) do
10:      DFS(ς′)
11:    end for
12:  else
13:    Report well-formed violation and exit
14:  end if
15: end procedure
```

6.2 Proof of correctness

Theorem 6 *A well-formed, terminating program can manifest a data-race on some schedule if and only if a computation graph generated by Algorithm 2 contains a data-race.*

Proof A program that is well-formed is guaranteed to not deadlock. Furthermore, it cannot introduce data-race via futures and Theorem 3 proves that all other determinism is impossible. If either of these properties is false, the theorem is vacuously true. When these properties are true, the model checker successfully performs a depth-first search on isolation orders, generating a computation graph for each possible order.

Since Algorithm 2 enumerates a computation graph for each possible ordering on **isolated** statements, each execution is compatible with one of these computation graphs. By Theorems 4 and 5, each of these graphs contains a data-race if and only if a compatible execution contains a data-race. □

7 Implementation and results

The model checking approach to data-race detection described in this paper has been implemented for Habanero Java. The implementation uses the verification run-time specifically designed to test Habanero Java programs and play nicely with JPF [1]. The implementation is a set of JPF listeners to create the computation graph and schedule over **isolated** statements. Two methods for data-race detection on the generated computation graph were tested. The first is an implementation of Algorithm 1. In this implementation, data-race is detected by intersecting the access sets of parallel nodes [28]. The second is an implementation of FastTrack [16], a

fast vector clock algorithm that optimizes the common case where only the clock value of the last thread to access a shared memory location is needed to determine whether a race condition exists. There are a few more key differences to the two methods. FastTrack checks for data-race alongside the execution of the program while the first method waits until program completion to check a graph for data-race.

This implementation uses JPF's VM listeners to track various program events related to parallelism. The methods `objectCreated` and `objectReleased` are used to create and connect nodes in the computation graph. The `objectCreated` method is used to track the creation of a new **async** task. It detects when a **async** statement executes and adds appropriate edges to the computation graph.

The `objectReleased` method is used to track when **finish** blocks complete execution and add edges as expected. The **await** statement is used to create a node in the graph where the tasks belonging to the **finish** block join. The `executeInstruction` method is used to track memory locations that are accessed by various tasks by updating the node with the location accessed by the task during the execution of that instruction. All in all, seven listeners and two factories are replaced in JPF consisting of roughly 1.6K lines of code.

The approach in this paper is compared to two other approaches implemented by JPF: *Precise race detector* (PRD) and *Gradual permission regions* (GPR). The PRD algorithm is a partial order reduction based on JPF's ability to dynamically detect shared memory accesses. In this mode, JPF schedules on all detected shared memory accesses. GPR uses program annotations to reduce the number of shared locations that need to consider scheduling by grouping several bytecodes that access shared locations into a single atomic block of code with read/write indications [31]. For example, if there are two bytecodes that touch shared memory locations, PRD schedules from each of the two locations. In contrast, if those two locations are wrapped in a single permission region, then GPR only considers schedules from the start of the region with the region being considered atomic. GPR is equal to PRD if every bytecode that accesses shared memory is put in its own region. Both approaches are a form of partial order reduction with GPR outperforming PRD by virtue of considering significantly fewer scheduling points via the user annotated permission regions.

The comparison over a set of benchmarks is shown in Table 1. The benchmarks are a collection of those from the Habanero Java distribution itself [1] and various presentation materials introducing the Habanero model; other benchmarks come from testing various language constructs in the development process. The table indicates for each benchmark its relative size in lines-of-code and tasks. The number of states generated by JPF for the proof, the time in minutes and seconds, and finally whether or not a race was detected. The "-" indicates that no results are available because the approach exceeded the arbitrary one hour time bound for each run. The "X" in the time column indicates that the analysis ran out of memory. Two times are reported for the approach in this paper, one for each method of data-race detection on the generated computation graphs: transitive closure (TC) and FastTrack (FT). The experiments were run on a machine with an Intel Core i5 processor with 2.8 GHz speed and 8 GB of RAM.

The table shows that in general, PRD does not finish in the time bound. The "Y*" on the **Race** column for PRD indicates that PRD incorrectly reports data-race on array objects in some examples. For GPR the "Y*" indicates that GPR reports the intended data-race between mutually exclusive regions. GPR falls behind quickly as the number of permission regions grow. The difference in performance is seen in the *Add*, *Scalar multiply*, and *Prime number counter* benchmarks which use shared variables. The regions are made as big as possible without creating a data-race. The last six benchmarks also have isolated regions. As a result, the state space for *computation graphs* is also large compared to other benchmarks. Of course, in the presence of isolation, the approach in this paper must enumerate all possible computation graphs, so it suffers the same state explosion as other model checking approaches.

On benchmarks with no data-race, the TC and FT algorithms are comparable. Because FT checks for data-race on the fly, it finishes for some benchmarks that have data-race while the TC algorithm does not. Even PRD and GPR finish on *True Dependant Linear* and *Last Private Missing* while the TC algorithm runs out of memory or times out. The TC algorithm can be made to be on the fly for programs without isolation by unioning the access sets of parent nodes and their direct children [28]. This adjustment would make it even more useful.

For each benchmark that caused the TC algorithm to time out or run out of memory, a smaller instance of the same benchmark was also tested (with the exception of *Jacobi Parallel*). These rows are marked with an asterisk in the benchmark name and is an example of how checking a program against a smaller problem instance can be a useful technique in data-race detection.

The benchmark *Many Network Requests* in an example of a case where the TC algorithm is more memory efficient than FT and therefore finishes while FT does not. In this benchmark, many tasks are spawned with few reads or writes per task. The benchmark is so named because it simulates a common idiom used in languages that support light weight tasks to initiate a network request with each task and then wait on the results at a later time. In this case, FT runs out

[1] https://wiki.rice.edu/confluence/display/HABANERO/Habanero-Java.

Table 1 Computation graphs versus permission regions versus PreciseRaceDetector

Test ID	SLOC	Tasks	***Computation graphs***				***GPR***			***PRD***		
			States	(TC) mm:ss	(FT) mm:ss	Race	States	mm:ss	Race	States	mm:ss	Race
Primitive array race	39	3	5	00:01	00:01	Y	5	00:00	Y	220	00:00	Y
Reciprocal array sum	58	2	4	00:03	00:03	N	32	00:06	Y	–	–	–
Primitive array No race	29	3	5	00:01	00:01	N	5	00:00	N	11,852	00:00	N
Two dim arrays	30	11	15	00:01	00:01	N	15	00:00	N	597	00:00	Y*
ForAll with iterable	38	2	9	00:01	00:01	N	9	00:00	N	–	–	–
Pipeline with futures	69	5	34	00:01	00:01	N	34	00:00	N	–	–	–
Add	67	3	11	00:02	00:01	N	62,374	00:33	N	4930	00:03	Y*
Scalar multiply	55	3	15	00:01	00:01	N	55,712	00:30	N	826	00:01	Y*
Vector Add	50	3	5	00:01	00:01	N	17	00:00	N	46,394	00:19	N
Clumped access	30	3	5	00:01	00:01	N	15	00:00	N	–	–	–
Anti dependant 1	29	1000	1006	00:36	00:02	Y	113	00:03	Y	23,881	00:53	Y
Anti dependant 2	32	1000	1006	00:36	00:02	Y	6	00:01	Y	24,189	00:58	Y
True dependant 1	27	1000	1006	00:37	00:01	Y	113	00:03	Y	23,188	00:55	Y
True dependant single element	28	1001	1007	00:35	00:02	Y	5	00:01	Y	–	–	–
True dependant linear	29	–	–	X	00:03	Y	232	00:04	Y	–	–	–
True dependant linear*	29	101	106	00:01	00:01	Y	71	00:01	Y	–	–	–
True dependant first dimension	34	–	–	–	12:01	Y	–	–	–	–	–	–
True dependant first dimension*	34	100	106	00:04	00:02	Y	9,520	00:15	Y	173,056	01:40	Y
Simd	32	101	106	00:01	00:01	Y	15	00:00	Y	–	–	–
Indirect access 1	73	181	186	00:03	00:02	Y	324	00:01	Y	–	–	–
Indirect access 2	73	181	186	00:03	00:03	Y	–	–	–	–	–	–
Indirect access 3	73	181	186	00:03	00:02	Y	–	–	–	–	–	–
Indirect access 4	73	181	186	00:03	00:02	Y	–	–	–	–	–	–
Minus minus	41	101	106	00:02	00:01	Y	11	00:01	Y	–	–	–
Simple simple simple	34	3	5	00:01	00:01	Y	4	00:01	Y	5,509	00:03	Y
3mm parallel	32	101	106	00:01	00:01	N	–	–	–	–	–	–
Last private missing	23	–	–	–	00:12	Y	4	00:10	Y	–	–	–
Last private missing*	23	101	106	00:02	00:01	Y	4	00:01	Y	3,951	00:04	Y
Do All 1	21	101	101	00:01	00:01	N	–	–	–	–	–	–

Table 1 continued

Test ID	SLOC	Tasks	*Computation graphs*				*GPR*			*PRD*		
			States	(TC) mm:ss	(FT) mm:ss	Race	States	mm:ss	Race	States	mm:ss	Race
Do All 2	23	101	106	00:03	00:02	N	–	–	–	–	–	–
Linear missing	42	101	106	00:02	00:01	Y	4	00:01	Y	4661	00:03	Y
Matrix multiply	33	–	–	–	X	–	–	X	–	–	X	–
Matrix multiply*	33	71	74	03:11	03:28	Y	–	X	–	–	X	–
Many network requests	24	1001	1007	00:29	X	N	–	–	–	–	–	–
Jacobi parallel	116	–	–	–	X	–	–	X	–	–	X	–
Isolated block No	39	4	28	00:01	00:01	N	8	00:01	Y*	–	–	–
Integer counter isolated	54	10	24	00:09	00:08	N	1,013,102	05:53	N	–	–	–
Data race isolate simple 1	46	2	9	00:01	00:01	Y	3	00:01	Y	182	00:01	Y
Data race isolate simple 2	53	3	10	00:01	00:01	Y	9	00:01	Y	480	00:01	Y
Prime num counter	51	25	75	00:01	00:01	N	3,542,569	17:37	N	–	–	–
Prime num counter ForAll	52	25	246	00:02	00:02	N	18	00:01	N	–	–	–
Prime num counter ForAsync	44	11	45,359	00:18	00:16	N	2,528,064	15:44	N	–	–	–

of memory quickly as it must maintain a vector clock whose size is linear in the number of tasks for each task and each shared variable. Neither algorithm finishes analyzing *Jacobi Parallel* but FT runs out of memory and TC times out.

The next set of results are for bigger *real-world* programs. The *Crypt-af* benchmarks is an implementation of the IDEA encryption algorithm and *Series-af* and *Series-f* are the Fourier coefficient analysis benchmarks adapted from the JGF suite [6] using **async-finish** and **future** constructs, respectively. The *strassen* benchmark is adapted from the OpenMP version of the program in the Kastors suite [38]. These are quickly verified free of data-race using computation graphs as shown below—PRD and GPR time out. Source code and additional benchmarks converted from benchmarks at https://github.com/LLNL/dataracebench can be found at https://bitbucket.org/byu-vv/jpf-hj/src/master/.

Test ID	SLOC	Tasks	States	(TC) mm:ss	(FT) mm:ss	Race
Crypt-af	1010	251	259	00:10	00:08	N
Series-af	730	99	106	00:03	00:01	N
Series-f	830	197	207	00:04	00:03	N
Strassen	560	1	2	0:37	00:37	N

8 Related work

Data-race detection in *unstructured thread parallelism*, where there is no defined protocol for creating and joining threads, or accessing shared memory, relies on static analysis to approximate parallelism and memory accesses [23,26,37] and then improves precision with dynamic analysis [10,12,16,18,27]. Other approaches reason about separate threads individually [19,29]. These approaches make few assumptions about the parallelism for generality and some are somewhat directly applicable to structured programs as seen in the prior section with the comparison to FastTrack [16].

There is newer more recent work to improve the efficiency and precision of data-race detection in unstructured thread parallelism. RacerD restricts the analysis to only report data-races for which it has high-confidence are real at the expense of proving a program data-race free [3]. Another approach compresses the execution trace using a grammar representation and analyzes the grammar for data-race [25]. It supports lock-set or happens-before analysis with vector clocks for detection. And yet another approach re-implements FastTrack and verifies the core algorithm correct, and thread safe, with CIVL [41]. None of these look specifically the structure of the parallelism *per se* and are best grouped with the techniques in the previous paragraph.

Structured parallelism constrains how threads are created and joined and how shared memory is accessed through programming models. For example, a locking protocol leads to static, dynamic, or hybrid lock-set analyses for data-race detection that are typically more efficient than approaches to unstructured parallelism [13,14,39]. It is worth noting that locking protocols can be applied to isolation with similar results—over-approximating the set of shared locations potentially rejecting programs as having data-race when indeed they do not.

Dynamic data-race detection based on *SP-bags* has been shown to effectively scale to large program instances and the method has been applied to the Habanero programming model to support a limited set of Habanero keywords including futures [34,35]. Another extension supports isolation [33] but is not complete, meaning it reports false races when isolated regions do not commute. The goal in this paper is verification and not run-time monitoring, so it needs to enumerate all possible computation graphs but can benefit from the more efficient SP-bags algorithm to detect data-race on-the-fly in the computation graph. Other dynamic data-race detectors for structured parallelism are based on access sets [28] and thread labels [30]. They can be more efficient for certain classes of programs but neither has been extended to support isolation.

Programmer annotations indicating shared interactions (e.g., permission regions) do improve model checking in general [40]. These are best understood as helping the partial order reduction by grouping several shared accesses into a single atomic block. The regions are then annotated with read/write properties to indicate what the atomic block is doing. The model checker only considers the interactions of these shared regions to reduce the number of executions explored to prove the system correct.

There are other model checkers for task-parallel languages [17,42]. The first modifies JPF and an X10 run-time extensively (beyond the normal JPF options for customization) and the second is a new virtual machine to model check the language. Both of these solutions require extensive programming whereas the solution in this paper leverages the existing Habanero verification run-time for JPF. That run-time maps tasks to threads making it small enough (relatively few lines of code) to argue correctness and making it work with JPF without any modification to JPF internals.

9 Conclusion and future work

This paper presents a model checking approach for data-race detection in task-parallel programs using computation graphs. The computation graph represents the happens-before relation of the task-parallel program and can readily be checked for data-race. The approach then enumerates all computation graphs created by different schedules of isolated regions to prove data-race freedom. The data-race detection analysis is implemented for a Java implementation of the Habanero programming model using JPF and evaluated on a host of benchmarks. The results are compared to JPF's precise race detector and a gradual permission regions based extension. The results show that computation graph analysis reduces the time required for verification significantly relative to JPF's standards.

The paper further implements FastTrack, and uses it in the model checking to compare its data-race detection performance on the computation graph with Algorithm 1 in this paper. To be clear, the model checking algorithm works with any data-race detection algorithm insofar as that algorithm is sound and complete with respect to the computation graph. The results over the benchmark set show that there are benefits to the on-the-fly nature of FastTrack, but that it is susceptible to overhead from the vector clocks. The results also confirm, in a small way, that it is often possible to instantiate small problem instances—at least in this benchmark set. It is hard to know how well that generalizes.

Future work is to reduce the number of schedules that must be considered by the model checker by weakening the happens-before relation in a manner similar to recent advances in dynamic data-race detection [20,24]. Soundly weakening the happens-before relation grows the number of schedules covered by any one observed program execution including schedules that have different orders of isolation statements. These larger equivalence classes captured by the weakened happens-before relation can be used to prune schedules from consideration by the model checker. Other future work is to leverage static analysis, abstract interpretation, etc., to reason over the input space so that the proof can be generalized to all inputs and executions.

References

1. Anderson P, Chase B, Mercer E (2014) JPF verification of Habanero Java programs. ACM SIGSOFT Softw Eng Notes 39(1):1–7
2. Bender MA, Fineman JT, Gilbert S, Leiserson CE (2004) On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In: Proceedings of the sixteenth annual ACM symposium on parallelism in algorithms and architectures, SPAA '04. ACM, New York, NY, USA, pp 133–144
3. Blackshear S, Gorogiannis N, Hearn PW, Sergey I (2018) RacerD: compositional static race detection. Proc ACM Program Lang 2(OOPSLA):144:1–144:28
4. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y (1996) Cilk: an efficient multithreaded runtime system. J Parallel Distrib Comput 37(1):55–69
5. Bouajjani A, Emmi M (2012) Analysis of recursively parallel programs. ACM SIGPLAN Not 47(1):203–214
6. Bull JM, Smith LA, Westhead MD, Henty DS, Davey RA (2000) A benchmark suite for high performance Java. Concurr—Pract Exp 12(6):375–388

7. Cavé V, Zhao J, Shirako J, Sarkar V (2011) Habanero-Java: the new adventures of old X10. In: Proceedings of the 9th international conference on principles and practice of programming in Java. ACM, pp 51–61
8. Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar Vivek (2005) X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not 40(10):519–538
9. Cheng G-I, Feng M, Leiserson CE, Randall KH, Stark AF (1998) Detecting data races in Cilk programs that use locks. In: Proceedings of the tenth annual acm symposium on parallel algorithms and architectures, SPAA '98. ACM, New York, NY, USA, pp 298–309
10. Choi J-D, Lee K, Loginov A, O'Callahan R, Sarkar V, Sridharan M (2002) Efficient and precise datarace detection for multithreaded object-oriented programs. SIGPLAN Not 37(5):258–269
11. Dennis JB, Gao GR, Sarkar V (2012) Determinacy and repeatability of parallel program schemata. In: Data-flow execution models for extreme scale computing (DFM), 2012, IEEE. IEEE, pp 1–9
12. Dimitrov D, Raychev V, Vechev M, Koskinen E (2014) Commutativity race detection. In: ACM SIGPLAN notices, vol 49:6. ACM, pp 305–315
13. Elmas T, Qadeer S, Tasiran S (2007) Goldilocks: a race and transaction-aware Java runtime. In: ACM SIGPLAN notices, vol 42:6. ACM, pp 245–255
14. Engler D, Ashcraft K (2003) RacerX: effective, static detection of race conditions and deadlocks. In: ACM SIGOPS operating systems review, vol 37:5. ACM, pp 237–252
15. Feng M, Leiserson CE (1997) Efficient detection of determinacy races in Cilk programs. In: Proceedings of the ninth annual acm symposium on parallel algorithms and architectures, SPAA '97. ACM, New York, NY, USA, pp 1–11
16. Flanagan C, Freund SN (2009) FastTrack: efficient and precise dynamic race detection. In: ACM sigplan notices, vol. 44:6. ACM, pp 121–133
17. Gligoric M, Mehlitz PC, Marinov D (2012) X10X: model checking a new programming language with an "old" model checker. In: 2012 IEEE fifth international conference on software testing, verification and validation (ICST), IEEE. IEEE, pp 11–20
18. Godefroid P (1997) Model checking for programming languages using verisoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '97. ACM, New York, NY, USA, pp 174–186
19. Gotsman A, Berdine J, Cook B, Sagiv M (2007) Thread-modular shape analysis. In: ACM SIGPLAN notices, vol 42:6. ACM, pp 266–277
20. Huang J, Meredith PO, Rosu G (2014) Maximal sound predictive race detection with control flow abstraction. SIGPLAN Not 49(6):337–348
21. Imam S, Sarkar V (2014) Habanero-Java library: a Java 8 framework for multicore programming. In: Proceedings of the 2014 international conference on principles and practices of programming on the Java platform: virtual machines, languages, and Tools. ACM, pp 75–86
22. Ji W (2013) TARDIS: task-level access race detection by intersecting sets
23. Kahlon V, Sinha N, Kruus E, Zhang Y (2009) Static data race detection for concurrent programs with asynchronous calls. In: Proceedings of the 7th joint meeting of the European software engineering conference and the foundations of software engineering. ACM, pp 13–22
24. Kini D, Mathur U, Viswanathan M (2017) Dynamic race prediction in linear time. SIGPLAN Not 52(6):157–170
25. Kini D, Mathur U, Viswanathan M (2018) Data race detection on compressed traces. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2018. ACM, New York, NY, USA, pp 26–37
26. Kulikov S, Shafiei N, Van Breugel F, Visser W (2010) Detecting data races with Java Pathfinder
27. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
28. Li L, Ji W, Scott ML (2014) Dynamic enforcement of determinism in a parallel scripting language. SIGPLAN Not 49(6):519–529
29. Malkis A, Podelski A, Rybalchenko A (2007) Precise thread-modular verification. In: Nielson HR, Filé G (eds) Static analysis. SAS 2007. Lecture notes in computer science, vol 4634. Springer, Berlin, Heidelberg, pp 218–232
30. Mellor-Crummey J (1991) On-the-fly detection of data races for programs with nested fork-join parallelism. In: Proceedings of the 1991 ACM/IEEE conference on supercomputing, supercomputing '91. ACM, New York, NY, USA, pp 24–33
31. Mercer E, Anderson P, Vrvilo N, Sarkar V (2015) Model checking task parallel programs using gradual permissions. In: Proceedings of 30th IEEE/ACM international conference on automated software engineering, new ideas category. ACM, pp. 535–540
32. Raman R, Zhao J, Sarkar V, Vechev M, Yahav E (2012) Scalable and Precise dynamic datarace detection for structured parallelism. In: ACM SIGPLAN notices, vol 47:6. ACM, pp 531–542
33. Raman R, Zhao J, Sarkar V, Vechev MT, Yahav E (2012) Efficient data race detection for async-finish parallelism. Form Methods Syst Des 41(3):321–347
34. Surendran R, Sarkar V (2016) Dynamic determinacy race detection for task parallelism with futures. In: 2016 16th international conference on runtime verification. Springer, Sprinter, pp 1–2
35. Surendran R, Sarkar V (2016) Dynamic determinacy race detection for task parallelism with futures. Springer International Publishing, Cham, pp 368–385
36. Utterback R, Agrawal K, Fineman JT, Lee I-T Angelina (2016) Provably good and practically efficient parallel race detection for fork-join programs. In: Proceedings of the 28th ACM symposium on parallelism in algorithms and architectures, SPAA '16. ACM, New York, NY, USA, pp 83–94
37. Vechev M, Yahav E, Raman R, Sarkar V (2011) Automatic verification of determinism for structured parallel programs. In: Cousot R, Martel M (eds) Static analysis. SAS 2010. Lecture notes in computer science, vol 6337. Springer, Berlin, Heidelberg, pp 455–471
38. Virouleau P, Brunet P, Broquedis F, Furmento N, Thibault S, Aumage O, Gautier T (2014) Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: 10th international workshop on OpenMP, IWOMP2014, IWOMP2014. Springer, Salvador, Brazil, France, pp 16–29
39. Voung JW, Voung Rx, Lerner S (2007) RELAY: static race detection on millions of lines of code. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, pp 205–214
40. Westbrook E, Zhao J, Budimlić Z, Sarkar V (2012) Practical permissions for race-free parallelism. In: ECOOP 2012–object-oriented programming. Springer, pp 614–639
41. Wilcox JR, Flanagan C, Freund SN (2018) VerifiedFT: a verified, high-performance precise dynamic race detector. SIGPLAN Not 53(1):354–367
42. Zirkel TK, Siegel SF, McClory T (2013) Automated verification of chapel programs using model checking and symbolic execution. NASA Form Methods 7871:198–212