Model-checking Task Parallel Programs for Data-race

Radha Nakade[1], Eric Mercer ✉[1], Peter Aldous[1], and Jay McCarthy[2]

[1] Brigham Young University, Provo, Utah
radha.nakade@gmail.com, {egm, aldous}@cs.byu.edu
[2] University of Massachusetts Lowell
jay.mccarthy@gmail.com

Abstract. Data-race detection is the problem of determining if a concurrent program has a data-race in some execution and input; it has been long studied and often solved. The research in this paper reprises the problem in the context of task parallel programs with the intent to prove, via model checking, the absence of data-race on any feasible schedule for a given input. Many of the correctness properties afforded by task parallel programming models such as OpenMP, Cilk, X10, Chapel, Habanero, etc. rely on data-race freedom. Model checking for data-race, presented here, is in contrast to recent work using run-time monitoring, log analysis, or static analysis which are complete or sound but never both. The model checking algorithm builds a happens-before relation from the program execution and uses that relation to detect data-race similar to many solutions that reason over a single observed execution. Unlike those solutions, model checking generates additional program schedules sufficient 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 existing data-race detectors in Java Pathfinder, show a significant reduction in the time required for proving data race freedom.

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 [1,2,3,4]. 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.

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 [5,6,7,8,9]; millions of tasks are feasible with varying degrees of slow-down (i.e., slow-down increases as parallelism constraints are relaxed) [10,11]. Other approaches use access histories [12,13] or programmer annotations [14]. Performance is a priority requiring careful integration into complex run-time environments, and solutions are only *complete*, meaning that nothing 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. Prior model checking based solutions enumerate schedules that interleave conflicting accesses, meaning at least one access is a write, to shared variables [15,16,17]. The approach here rather uses techniques from dynamic approaches to build a happens-before relation 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 [18,19,20]. As such, the happens-before relation captures in one many of the schedules exhaustively enumerated by the prior model checking solutions. Unlike the dynamic approaches, though, the approach here then 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 [15,16,17]. 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. Of course, as with any model checking

```
proc m (var x : int)
    g := 0;
    post r ← p 0 λv. skip;
    [ 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)

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

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 tens of mutually exclusive regions. It further assumes that a data-race freedom proof on the small instance generalizes to the large instance. The primary contributions are thus

– a simple approach to data-race detection in programs that terminate based on constructing 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 results from benchmarks comparing to other solutions in JPF.

The rest of this paper is organized as follows: Sec. 2 illustrates the approach in a small example; Sec. 3 defines the computation graph, task parallel programs, and how to build a computation graph from a program execution; Sec. 4 gives a correctness proof; Sec. 5 is the empirical study with a summary of the implementation; and Sec. 7 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. 1(a). 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. 1(b) is the equivalent program in the Habanero Java language.

For Fig. 1, execution begins with the procedure `m`. The variable `g` is global. The **post**-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 the λ-expression as a return value handler. In this case, that handler is 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 at its **return**-statement. The expression in the **return**-statement is evaluated at the join and the value 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 any unprotected shared memory access leading quickly to state explosion [15,16,17]. 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, [18], but is easily extended to something like weak causally-precedes to further reduce checked schedules [19,20]. 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, during execution, the happens-before relation in the form or a computation graph. The left part of Fig. 2 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 `p`-labeled line is the result of the **post**-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. Such a graph can be readily checked for data-race in linear time [18].

To reason over all schedules, the approach in this paper first assumes two restrictions common in most task parallel languages: if a return value handler side-effects, then it exists in a region by itself throughout its lifetime, and all tasks are joined at termination in a deterministic order by a implicit enclosing parent task. Under these restrictions, the model checker, to prove data-race freedom, must generate a set of schedules that contains all ways allowed by the

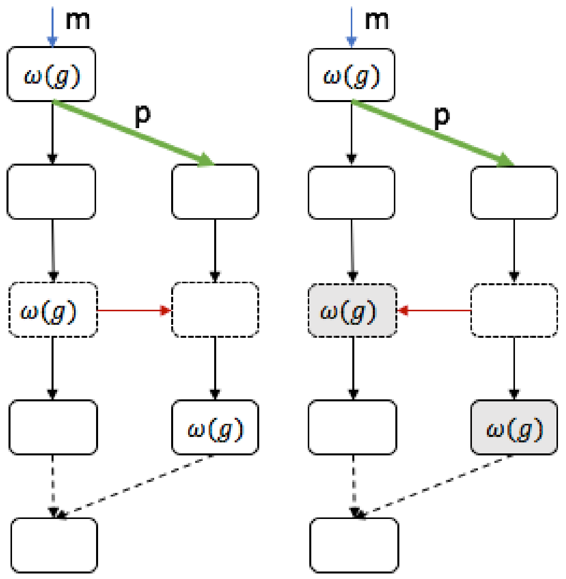

Fig. 2. Two computation graphs for Fig. 1: no data-race on left and data-race right.

program semantics to interleave **isolated**-statements. This result is the main contribution.

The right part of Fig. 2 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 programs that can be instantiated on small problem instances; however, in general, a proof on a small problem instance typically generalizes to large problem instances.

3 Task Parallel Programs

A task parallel program uses a general programming model to define parallelism. That model, under certain restrictions, captures the semantics of many common task parallel models such as Habanero, X10, Cilk, OpenMP, Chapel, etc. This section first defines the structure of the computation graph, and it then discusses task parallel programs and how a computation graph can be created from an observed execution of a task parallel program. The full semantics for the programming model, and how the computation graph is formally constructed from an execution of the program, is not presented here but can be found at `https://jpf.byu.edu/jpf-hj/`.

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 [21]. It is modified here to track memory locations accessed by tasks.

Definition 1. *A computation graph is 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,* $\omega : (N \mapsto \mathcal{P}(\texttt{Globals}))$ *maps* N *to the unique identifiers for the shared locations written by the tasks, and* `Globals` *is the set of the unique identifiers for the shared locations.*

The graph captures the happens-before relation between nodes: $\prec \subset N \times N$. 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., $n_i \not\prec n_j \land n_j \not\prec n_i$ or, equivalently, $n_i \mid\mid_{\prec} n_j$), and the two nodes conflict:

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

The condition is readily checked in linear time as mentioned in the previous section.

3.2 Programming Model

The programming model is derived from Bouajjani and Emmi for isolated parallel tasks [22]; this variant removes the isolation between tasks with the introduction of shared memory. It additionally restricts task passing to only allow tasks to be passed when a child completes, and those tasks are only passed to the parent. Finally, procedures with side-effecting return value handlers must be the only members of their respective regions.

The surface syntax for the language is given in Fig. 3. A program **P** is a sequence of procedures. Each procedure has a single parameter `l` of type L. Procedures may also reference shared variables taken from a finite set of names which include a special reserved variable `isolate` that is only used by the semantics for mutual exclusion. The body of a procedure is inductively defined by **s**. The expression language, e, is also abstracted.

The **post**, **await**, **ewait**, and **isolated** statements relate to concurrency and affect the shape of the computation graph; the rest of the statements have their usual sequential meaning. The semantics produce a computation graph as a by-product of reducing the program via rewrites. Two additional data are associated with the computation graph and are used by the semantics in the construction: *last* is a special node used to assert the observed order of **isolated**-statements and $R : \texttt{Regs} \mapsto \mathcal{P}(N)$ is a function used to join tasks in a region at synchronization. In general, a function notation is adopted to access members

$$
\begin{array}{rcl}
\mathbf{P} & ::= & (\mathbf{proc}\ p\ (\mathbf{var}\ \mathtt{l} : L)\ s)* \\
\mathbf{s} & ::= & s;\ s \mid \mathtt{l} := e \mid \mathbf{skip} \mid [\mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ s] \\
& \mid & [\mathbf{while}\ e\ \mathbf{do}\ s] \mid \mathbf{call}\ \mathtt{l}\ := p\ e \mid \mathbf{return}\ e \\
& \mid & \mathbf{post}\ r \leftarrow p\ e\ d \mid \mathbf{await}\ r \mid \mathbf{ewait}\ r \\
& \mid & [\mathbf{isolated}\ s]
\end{array}
$$

Fig. 3. The surface syntax for task parallel programs.

of tuples. For example, the members of the $G = (N, E, \rho, \omega, \mathit{last}, R)$ are accessed as $N(G)$, $E(G)$, $\rho(G)$, etc.

The **post**-statement adds a task into a region r, taken from a finite set of region identifiers, by indicating the procedure p for the task with an expression for the local variable value e, and a return value handler d to run in the context of the parent task. The Post rewrite rule adds two fresh nodes n_0' and n_1 to the computation graph: node n_0' represents the statements following **post** and n_1 represents the statements to be executed by the new task: $N(G') = N(G) \cup \{n_0', n_1\}$ and $E(G') = E(G) \cup \{(n_0, n_0'), (n_0, n_1)\}$. 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, $\eta(e)$, for the local parameter value in the new task: $\rho(G') = \rho(G)[n_0 \overset{\cup}{\mapsto} \eta(e, \sigma)]$; meaning that the ρ function is as before only now it additionally includes the variables read by $\eta(e, \sigma)$ in the read set for n_0—σ in the store value lookup.

The **await** and **ewait** statements synchronize a task with the sub-ordinate tasks in the indicated region. Intuitively, when a task calls **await** on region r, it is blocked until all the tasks it owns in r finish execution. Similarly, when a task issues an **ewait** with region r, it is blocked until one task it owns in r completes. A task is termed *completed* when its statement is a **return**-statement.

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, say its t_2, in the region is non-deterministic; 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, $n(t_2)$, to join later once the region is empty, $R(G') = R(G)[r \overset{\cup}{\mapsto} n(t_2)]$, and it updates the read set for t_2 on the expression in the **return**-statement: $\rho(G)[n(t_2) \overset{\cup}{\mapsto} \eta(e, \sigma)]$. 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 (e.g., the new statement context replaces $S[\mathbf{await}\ r]$ with $S[s;\ \mathbf{await}\ r]$ where s in the return value handler).

The Await-Done rule activates when the last task, let us call it t_2, in the region is joined. It differs from the Await rule in that it constrains 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 since the region is now empty: $n' = \text{fresh}()$, $N(G') = N(G) \cup \{n'\}$, and $E(G') = E(G) \cup \{(n, n'), (n(t_2), n')\} \cup \{(n_i, n') \mid n_i \in R(G)(r)\}$

The EWAIT and EWAIT-DONE rules follow AWAIT and AWAIT-DONE respectively only without the recursive statement when the region is not empty since it only needs to wait on a single task to complete. The rules delay the ordering of tasks joined in the region to when the region becomes empty (i.e., the last task joins) just as done for **await**-statements.

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.

If no other isolated statements are running, then the ISOLATED rule updates the `isolated` shared variable to block other tasks from isolating and inserts after the isolated statement s the new **isolated-end** keyword to reset the shared variable at the end of isolation. 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: $n' = \text{fresh}()$, $N(G') = N(G) \cup \{n'\}$, and $E(G') = E(G) \cup \{(n, n'), (\mathit{last}(G), n')\}$. As a note, *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. **isolated**-statements are totally ordered in the computation graph: $n' = \text{fresh}()$, $\mathit{last}(G') = n$, $N(G') = N(G) \cup \{n'\}$, and $E(G') = E(G) \cup \{(n, n')\}$.

4 Proof of Correctness

For a given program and input, the computation graphs produced by the tree semantics summarized in Section 3.1 demonstrate a data race if and only if a data race is possible for the given program and input. Before proving this claim, we formally define data race. The definition of data race depends on definitions for concurrency and conflict.

Definition 2 (*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.*

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

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

Definition 3 (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.*

Without loss of generality, we call the program's initial state ς_0.

Definition 4 (Concurrency). *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:*

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

A state ς that satisfies this condition for s and s' is called a witness state for $s \mid\mid s'$. $\overset{*}{\to}$ is the transitive closure of the transition relation $\to$ defined in the semantics.

Definition 5 (Data race). *There is a data race on two statements s and s' if and only if they conflict and are concurrent:*

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

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. The semantics ensure that no two statements inside of **isolated**-statements can be concurrent, as only one thread may enter an **isolated**-statement at a time.

Before proving the correctness of data races in the computation graph, we observe that only nodes that end in a **post**-statement and nodes for **isolated**-statements can have multiple outgoing edges. In both cases, the nodes reached by these edges are all in distinct threads of execution. Similarly, only **isolated**-statements and nodes following an **await**- or **ewait**-statement (in the parent thread) and following **return**-statements (in child threads) have multiple incoming edges. The edges that converge on these nodes come from distinct threads.

Lemma 1. *If two nodes n and n' are unordered in a state's computation graph G, every $s \in n$ and $s' \in n'$ are concurrent:*

$$\forall s \in n, s' \in n' : n \mid\mid_{\prec} n' \implies s \mid\mid s'. \tag{5}$$

Proof. The two nodes n and n' are both reachable; otherwise, they would not have been generated in G. Within a node, it is possible to advance or wait independent of other nodes' behaviors. Accordingly, it is possible to begin at ς_0 and advance until s is active. Similarly, it is possible to advance until s' is active. What remains to be proven is whether or not it is possible to reach a state where both s and s' are active; in other words, if it is possible for some schedule to reach s in one task and then to reach s' in some other task without advancing the first task any further.

By the construction of G, n and n' must have some least common ancestor n_A that is also reachable. n_A must either end in a **post**-statement or be an **isolated**-statement, as the reduction rules only allow these two statements to have multiple outgoing edges. In both cases, the child nodes of n_A must be in different tasks. Without loss of generality, we say that t either contains n or is some ancestor of the task that does. Similarly, we say that t' either contains n' or is an ancestor of the task that does.

We first advance to n_A on some schedule that does not contradict $\prec$. This is possible because n_A, n, and n' were all generated. Execution may block, necessitating the advancement of other threads; however, no relationship can exist between the thread that leads to n', as this would contradict the definition of n_A as least common ancestor. We advance t and any relevant children or unrelated threads until reaching n and then proceed until reaching s. Then, we advance t' and any relevant children or unrelated threads until reaching n' and then proceed until reaching s'.

Both s and s' are active, so they are concurrent.

Lemma 2 (Soundness of *conflict* over nodes). *If two nodes conflict, there exists a pair of statements, one from each node, that conflicts:*

$$\mathit{conflict}\,(n, n') \implies \exists s \in n, s' \in n' : \mathit{conflict}\,(s, s')\,. \tag{6}$$

Proof. If two nodes conflict, it is because ρ and ω were updated in some reduction. More specifically, if $\rho\,(n) \neq \emptyset$, at least one statement $s \in n$ must read a global variable; the reduction of statements that read a global variable are the only way that ρ updates. The same reasoning applies to ω.

Lemma 3 (Completeness of *conflict* over nodes). *If two statements conflict, their respective nodes will conflict.*

$$\forall s \in n, s' \in n' : \mathit{conflict}\,(s, s') \implies \mathit{conflict}\,(n, n')\,. \tag{7}$$

Proof. If $s \in n$, then s must have been reduced in n. Per the semantics, $\rho\,(n)$ and $\omega\,(n)$ must be updated to include reads and writes from s. Accordingly, nodes conflict whenever statements they include conflict.

Theorem 1 (Soundness of computation graph over data races). *If a computation graph reports a data race, there is a data race in the program on the given input.*

Proof. By Lemma 1 and Lemma 2.

Theorem 2 (Completeness of computation graph over data races). *If there is a data race in the program on the given input, a computation graph generated by the model checker reports a data race.*

Proof. We choose, without loss of generality, the first data race manifest along some schedule in the program. Our algorithm reports the first data race it finds and exits. This is consistent with the theorem definition.

The definition of data race states that the two statements must be concurrent, which implies that it is possible to generate a witness state for the two statements' concurrency. As a result, any two statements that conflict and are both reachable will be members of nodes in some computation graph. Because the model checker generates all possible computation graphs, it generates the computation graph created by the witness state. By Lemma 1 and Lemma 3, this computation graph demonstrates the data race.

5 Implementation and Results

The model checking approach to data race detection described in this paper has been implemented for Habanero Java (HJ). The implementation uses the verification run-time specifically designed to test HJ programs and play nicely with JPF [17]. The implementation is a set of JPF listeners to create the computation graph and only schedule on **isolated**-statements. It is worth noting that this implementation does not use vector clocks for data-race detection on the generated computation graph but rather uses a more direct, albeit naive, quadratic check via transitive closure.

As a sketch of the implementation, JPF's VM listeners are used to track various program events related to parallelism. The methods `objectCreated` and `objectReleased` are used to create nodes in the computation graph. The `objectCreated` method is used to track the creation of a new **async** task. The `objectCreated` method detects when a **post** statement executes and adds appropriate edges to the computation graph. Similarly, the `objectReleased` method is used to track when **finish** blocks complete execution. 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 [23]. 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 HJ distribution itself [3] 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 experiments were run on a machine with an Intel Core i5 processor with 2.6GHz speed and 8GB of RAM.

Table 1. Computation graphs vs Permission Regions vs. PreciseRaceDetector.

			Computation graphs			***GPR***			***PRD***		
Test ID	**SLOC**	**Tasks**	**States**	**mm:ss**	**Race**	**States**	**mm:ss**	**Race**	**States**	**mm:ss**	**Race**
Primitive Array Race	39	3	5	00:00	Y	5	00:00	Y	220	00:00	Y
Reciprocal Array Sum	58	2	4	00:08	Y	32	00:06	Y	-	-	-
Primitive Array No Race	29	3	5	00:00	N	5	00:00	N	11,852	00:00	N
Two Dim Arrays	30	11	15	00:00	N	15	00:00	N	597	00:00	Y*
ForAll With Iterable	38	2	9	00:00	N	9	00:00	N	-	-	-
Integer Counter Isolated	54	10	24	00:01	N	1,013,102	05:53	N	-	-	-
Pipeline With Futures	69	5	34	00:00	N	34	00:00	N	-	-	-
Prime Num Counter	51	25	776	00:01	N	3,542,569	17:37	N	-	-	-
Prime Num Counter ForAll	52	25	30	00:02	N	18	00:01	N	-	-	-
Prime Num Counter ForAsync	44	11	653	00:01	N	2,528,064	15:44	N	-	-	-
Add	67	3	11	00:01	N	62,374	00:33	N	4930	00:03	Y*
Scalar Multiply	55	3	15	00:01	N	55,712	00:30	N	826	00:01	Y*
Vector Add	50	3	5	00:00	N	17	00:00	N	46,394	00:19	N
Clumped Access	30	3	5	00:03	N	15	00:00	N	-	-	-

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 because it does not check the indexes—a shortcoming in the PRD implementation. 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 *Prime number counter* benchmark also has isolated sections. 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.

The next set of results are for bigger *real-world* programs. The *Crypt-af* and *Crypt-f* benchmarks are implementations of the IDEA encryption algorithm and

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

Series-af and *Series-f* are the Fourier coefficient analysis benchmarks adapted from the JGF suite [24] using **async-finish** and **future** constructs respectively. The *strassen* benchmark is adapted from the OpenMP version of the program in the Kastors suite [25]. 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 `https://github.com/LLNL/dataracebench` can be found at `https://jpf.byu.edu/jpf-hj/`.

Test ID	**SLOC**	**Tasks**	**States**	**mm:ss**	**Race**
Crypt-af	1010	259	260	00:17	N
Crypt-f	1145	387	775	00:46	N
Series-af	730	329	750	00:36	N
Series-f	830	354	630	00:51	N
Strassen	560	3	7	00:57	N

6 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 [26,27,28] and then improves precision with dynamic analysis [18,29,30,31,32]. Other approaches reason about threads individually [33,34]. These approaches make few assumptions about the parallelism for generality and typically have higher cost for analysis. It is difficult to compare the approach in this paper to these more general approaches because the work in this paper relies critically on the structure of the parallelism to reduce the cost of formal analysis.

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 [35,36,37]. 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 but not isolation [10,11]. 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.

Programmer annotations indicating shared interactions (e.g., permission regions) do improve model checking in general [14]. 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 [15,16]. 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 runtime 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.

7 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.

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 [19,20]. 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, to reason over the input space so that the proof can be generalized to all inputs and executions.

Acknowledgement.

This work is supported by the National Science Foundation under grant 1302524. Thanks to Lincoln Bergeson and Kyle Storey for their contribution to the tool and its tests. The final authenticated version is available online at `https://doi.org/10.1007/978-3-319-77935-5_25`.

References

1. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing **37**(1) (1996) 55–69
2. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing. SIGPLAN Not. **40**(10) (October 2005) 519–538
3. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the new adventures of old X10. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, ACM, ACM, ACM (2011) 51–61
4. Imam, S., Sarkar, V.: 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, ACM, ACM (2014) 75–86
5. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in Cilk programs. In: Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA '97, New York, NY, USA, ACM (1997) 1–11
6. Cheng, G.I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F.: Detecting data races in Cilk programs that use locks. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA '98, New York, NY, USA, ACM (1998) 298–309
7. Bender, M.A., Fineman, J.T., Gilbert, S., Leiserson, C.E.: 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, New York, NY, USA, ACM (2004) 133–144
8. Raman, R., Zhao, J., Sarkar, V., Vechev, M.T., Yahav, E.: Efficient data race detection for async-finish parallelism. Formal Methods in System Design **41**(3) (2012) 321–347
9. Utterback, R., Agrawal, K., Fineman, J.T., Lee, I.T.A.: 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, New York, NY, USA, ACM (2016) 83–94
10. Surendran, R., Sarkar, V.: Dynamic Determinacy Race Detection for Task Parallelism with Futures. In: Runtime Verification, 2016 16th International Conference on, Springer, Springer, Sprinter (2016) 1–2
11. Surendran, R., Sarkar, V. In: Dynamic Determinacy Race Detection for Task Parallelism with Futures. Springer International Publishing, Cham (2016) 368–385
12. Mellor-Crummey, J.: 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, New York, NY, USA, ACM (1991) 24–33
13. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Scalable and Precise dynamic datarace detection for structured parallelism. In: ACM SIGPLAN Notices. Volume 47:6., ACM, ACM, ACM (2012) 531–542
14. Westbrook, E., Zhao, J., Budimlić, Z., Sarkar, V.: Practical permissions for race-free parallelism. In: ECOOP 2012–Object-Oriented Programming. Springer, Springer (2012) 614–639
15. Gligoric, M., Mehlitz, P.C., Marinov, D.: X10X: Model checking a new programming language with an "old" model checker. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, IEEE, IEEE, IEEE (2012) 11–20

16. Zirkel, T.K., Siegel, S.F., McClory, T.: Automated Verification of Chapel Programs using Model Checking and Symbolic Execution. NASA Formal Methods **7871** (2013) 198–212
17. Anderson, P., Chase, B., Mercer, E.: JPF verification of Habanero Java programs. ACM SIGSOFT Software Engineering Notes **39**(1) (2014) 1–7
18. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (1978) 558–565
19. Kini, D., Mathur, U., Viswanathan, M.: Dynamic race prediction in linear time. SIGPLAN Not. **52**(6) (June 2017) 157–170
20. Huang, J., Meredith, P.O., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. SIGPLAN Not. **49**(6) (June 2014) 337–348
21. Dennis, J.B., Gao, G.R., Sarkar, V.: Determinacy and repeatability of parallel program schemata. In: Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012, IEEE, IEEE, IEEE (2012) 1–9
22. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. ACM SIGPLAN Notices **47**(1) (2012) 203–214
23. Mercer, E., Anderson, P., Vrvilo, N., Sarkar, V.: Model checking task parallel programs using gradual permissions. In: Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering, New ideas category, ACM, ACM, ACM (2015) 535–540
24. Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S., Davey, R.A.: A benchmark suite for high performance Java. Concurrency - Practice and Experience **12**(6) (2000) 375–388
25. Virouleau, P., BRUNET, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP Dependent Tasks with the KASTORS benchmark suite. In: 10th International Workshop on OpenMP, IWOMP2014. 10th International Workshop on OpenMP, IWOMP2014, Salvador, Brazil, France, Springer (September 2014) 16 – 29
26. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: Proceedings of the the 7th joint meeting of the European software engineering conference and The foundations of software engineering, ACM (2009) 13–22
27. Kulikov, S., Shafiei, N., Van Breugel, F., Visser, W.: Detecting data races with Java Pathfinder (2010)
28. Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In: Static Analysis. Springer, Springer (2011) 455–471
29. Godefroid, P.: Model checking for programming languages using verisoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '97, New York, NY, USA, ACM (1997) 174–186
30. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: ACM Sigplan Notices. Volume 44:6., ACM, ACM, ACM (2009) 121–133
31. Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. SIGPLAN Not. **37**(5) (May 2002) 258–269
32. Dimitrov, D., Raychev, V., Vechev, M., Koskinen, E.: Commutativity race detection. In: ACM SIGPLAN Notices. Volume 49:6., ACM, ACM, ACM (2014) 305–315
33. Malkis, A., Podelski, A., Rybalchenko, A.: Precise thread-modular verification. In: Static Analysis. Springer, Springer (2007) 218–232

34. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: ACM SIGPLAN Notices. Volume 42:6., ACM, ACM, ACM (2007) 266–277
35. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: ACM SIGOPS Operating Systems Review. Volume 37:5., ACM, ACM, ACM (2003) 237–252
36. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware Java runtime. In: ACM SIGPLAN Notices. Volume 42:6., ACM, ACM, ACM (2007) 245–255
37. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, ACM, ACM (2007) 205–214