Model-checking Task Parallel Programs for Data-races using Computation Graphs[*]

Radha Nakade and Eric Mercer
Brigham Young University
Provo, UT
egm@cs.byu.edu

Jay McCarthy
University of Massachusetts Lowell
Lowell, Massachusetts
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 for different contexts and goals. The research in this paper reprises the problem of data-race detection but in the context of model checking as opposed to run-time monitoring or static analysis. The programming model is task parallel where computations are hierarchically divided into non-isolated parallel executing tasks and is suitable to describing real-world languages (i.e., Cilk, X10, Chapel, Habanero, etc.). The model semantics are defined to construct a computation graph and a naive algorithm to detect data-race on a computation graph is given. A fragment of the programming model is defined such that the algorithm becomes sound and complete for computation graphs from a single observed program execution. Model checking is applied to programs outside this fragment to enumerate all possible computation graphs. 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 data race detection.

CCS CONCEPTS

• **Software and its engineering → Formal methods**;

KEYWORDS

Task Parallel Languages, Data-race, Model Checking, Computation Graphs

ACM Reference format:
Radha Nakade and Eric Mercer and Jay McCarthy. 2017. Model-checking Task Parallel Programs for Data-races using Computation Graphs. In *Proceedings of SPIN Symposium on Model Checking Software, Santa Barbara, California USA, July 2017 (SPIN'2017),* 10 pages.
DOI:

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 problem of *Data-race detection*, 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 data-race detection 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., Cilk, X10, Chapel, Habanero, etc.). These models rely on run-time environments to implement task abstractions to represent concurrent executions [4, 9, 10, 27]. 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 are predicated on the input programs being data-race free, 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. The predominant *SP-bags* algorithm, with its variants, exploits assumptions on task creation and joining for efficient on-the-fly detection with low overhead [3, 11, 19, 40, 47]; millions of task are feasible with varying degrees of slow-down (i.e., slow-down increases as parallelism constraints are relaxed) [43, 44]. Other approaches use access histories [35, 39] or programmer annotations [51, 52]. Performance is a priority, and many solutions are only *complete*, meaning that nothing can be concluded about other executions of the same program.

The research presented in this paper reprises data-race detection in task parallel models in the context of model checking under two assumptions: first, data-race is largely independent of the size of the problem instance; and second, it is possible to instantiate small problem instances. These assumptions are indeed implicit in other model checking solutions for task parallel models—a small problem instance is the best solution to state explosion [2, 22, 56]. Prior approaches, however, extensively modify the language run-time and the model checker. Such solutions require source code [22], sometimes require the user to specify the number of processors modeled making it difficult to generalize [56], or rely on user annotations to indicate sharing so data-race may be missed [2]. The solution here is to make clear the requirements on the run-time, use semantics that are independent of actual hardware, and automatically detect data-race without annotations.

[*]This research is funded by NSF Grant 1302524

The approach first defines a *computation graph* to abstractly model parallelism with a naive algorithm to detect data-race. Computation graph construction is then formally defined in a general task parallel model based on partitioning concurrent executions into hierarchical regions with shared locations. Such a model is suitable to describe real-world languages (e.g., Cilk, X10, Chapel, Habanero, etc.). A fragment of the model is then defined so that data-race detection on a computation graph from a single execution is both sound and complete (e.g., and single trace is sufficient to prove the absence of data-race in all execution schedules). Any approach that only allows programs in the deterministic fragment of the language is able to make similar guarantees. The deterministic fragment is then expanded to show how model checking may be applied to enumerate the space of computation graphs for data-race detection. Finally, the approach is evaluated on a Java implementation of Habenero with the Java Pathfinder model checker (JPF). 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 computation graph approach to be more efficient in JPF terms with its overhead. The primary contributions are thus

- a simple approach to data-race detection based on constructing a computation graph from an execution of a task parallel program;
- the computation graph construction in terms of general semantics suitable for real-world languages;
- a scheduling algorithm for model checking when a programmer uses mutual exclusion (e.g., a way to enumerate all possible computation graphs); and
- an implementation of the approach for Java Habanero in JPF with results from benchmarks comparing to other solutions in JPF.

2 DATA RACE DETECTION

We start with a computation graph in the presentation because the goal is to build a computation graph from a program execution, and then analyze that graph for data-race. If a program is deterministic, then only a single execution (and single graph) is needed to prove or disprove data-race. If a program is non-deterministic due to mutual exclusion, then all possible computation graphs must be considered. Each of these items is discussed after the computation graph construction is defined in the next section.

A Computation Graph for a task parallel program is a directed acyclic graph that represents the execution of the program [14]. It is modified here to track memory locations accessed by tasks.

Definition 2.1. **Computation Graph:** A Computation Graph $G = \langle N, E, \delta, \omega \rangle$ of a task parallel program **P** with input ψ is a directed acyclic graph where N is a finite set of nodes, $E \subseteq N \times N$ is a set of directed edges, δ is the function that maps N to the unique identifiers for the shared locations read by the tasks: $\delta : (N \mapsto 2^V)$, ω is the function that maps N to the unique identifiers for the shared locations written by the tasks: $\omega : (N \mapsto 2^V)$, and V is the set of the unique identifiers for the shared locations.

Fig. 1 is simple computation graph with nodes. Every node represents a block of sequential operations and edges order the nodes. The order between any two nodes n_1 and n_2 is given as $n_1 \prec n_2$, meaning that n_1 happens before n_2. Parallel nodes are un-ordered: $n_1 \nprec n_2$ and $n_2 \nprec n_1$. Data-race detection finds un-ordered nodes with conflicting accesses. For the example, nodes n_1 and n_2 are unordered and both are writing to variable r_1.

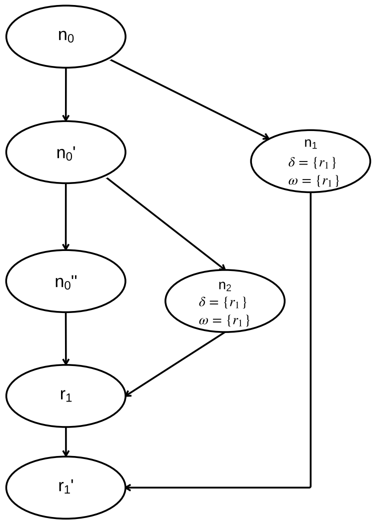


Figure 1: Computation Graph Example.

Algorithm 1 Data Race detection in a computation graph

```
 1: function DetectRace(ComputationGraph G)
 2:     N := Topologically ordered nodes in G
 3:     for i in [1, |N|] do
 4:         n = N[i]
 5:         for j in [i+1, |N|] do
 6:             n' := N[j]
 7:             if (n ⊀ n') then
 8:                 if ((δ(n) ∩ ω(n') ≠ ∅) ∨ (ω(n) ∩ δ(n') ≠ ∅) ∨ (ω(n) ∩ ω(n') ≠ ∅)) then
 9:                     Report Data Race and Exit
10:                 end if
11:             end if
12:         end for
13:     end for
14: end function
```

A naive approach to data-race detection given a computation graph is in Algorithm 1. The nodes in the computation graph are added to a topologically sorted set on Line 2. The i^{th} node in the set is given by $N[i]$. The nodes are traversed in order and each node is compared to every node that comes later in the topological ordering. Line 7 checks if the nodes n and n' are un-ordered. If the nodes are un-ordered, then the sets of memory locations accessed by each node are checked for conflict on Line 8. If any of the sets shares an element, then there is a data race.

The algorithm is purposely naive and can be improved if needed [35, 39]; though, it is not the goal of the research presented here because problem instances are assumed to be small enough for model checking which is limited by the number of computation graphs that must be enumerated rather than the size of those graphs.

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

Figure 2: The surface syntax for task parallel programs.

Theorem 2.2. *Using the tree semantics with Algorithm 1 to detect data-race in the resulting computation graph is complete for a task parallel program with a given input.*

3 COMPUTATION GRAPH CREATION

This section formally defines how to build a computation graph from an execution trace of a task parallel program. The formalism is driven by a desire to be general enough to cover any language feature in a task parallel language while being specific enough to precisely define the graph construction. The programming model is derived from Bouajjani and Emmi for isolated parallel tasks [5]. The derivative in this paper allows sharing since programmers use it (intentionally or otherwise). The definition is thus expanded from the original in two ways: first, parallel regions now include a single shared variable that can be accessed by any task with a handle to the region; and second, tasks include additional sets indicating which regions are for reading and which regions are for writing relative to the shared region variable in each region.

The surface syntax for the language is given in Fig. 2. A program **P** is a sequence of procedures. The procedure name p is taken from a finite set of names `Proc`. Each procedure has a single L-type parameter `l` taken from a finite set of parameter names `Vars`. The body of the procedure is inductively defined by s. The semantics is abstracted over concrete values and operations, so the possible types of `l` are not specified nor is the particular expression language, e, but assume it includes variables references and Boolean values (**true** and **false**). The details of either L or e are never relevant for computation graph construction and are thus omitted. The set of all expressions is given by `Exprs`. Values are given by the finite set `Vals` and include at least Boolean values. `Exprs` contain `Vals` and the *choice operator* $\star$.

The statements (s) of the language denote the behavior of the procedure. Most statements, like the **if**-statement, **;**-statement, and **while**-statement have their typical meaning. Other statements require further explanations.

Statements are divided into the concurrent statements (**post**-statement, **await**-statement and **ewait**-statement) and sequential statements (everything else). Let `Regs` be a finite set of region identifiers. Associated with each region r is a single variable referenced in the surface syntax by $\mathtt{l}(r)$. A task is posted into a region r by indicating the procedure p for the task with an expression for the local variable value e, three lists of regions from `Regs`* (i.e., the Kleene closure on `Regs`), and a return value handler d. For the region lists, $\vec{r}$ are regions whose ownership is transferred from the parent to the new child task (i.e., the child now owns the tasks in those regions),

$$
\begin{array}{l}
\textbf{proc}\ \text{main}\ (\textbf{var}\ \mathtt{n}\ :\ \textbf{int}) \\
\quad \mathtt{n}\ :=\ \mathtt{1}; \\
\quad \textbf{post}\ r_1 \leftarrow p_1\ n\ \varepsilon\ (r_1)\ (r_1)\ \lambda v.n := n + v; \\
\quad \textbf{post}\ r_1 \leftarrow p_2\ n\ \varepsilon\ (r_1)\ (r_1)\ \lambda v.n := n + v; \\
\quad \textbf{await}\ r_1 \\
\textbf{proc}\ p_1\ (\textbf{var}\ \mathtt{n} : \textbf{int}) \\
\quad \mathtt{l}(r_1) := \mathtt{l}(r_1) + n; \\
\quad \textbf{return}\ (\mathtt{n}\ +\ \mathtt{1}) \\
\textbf{proc}\ p_2\ (\textbf{var}\ \mathtt{n} : \textbf{int}) \\
\quad \mathtt{l}(r_1) := \mathtt{l}(r_1) + n; \\
\quad \textbf{return}\ (\mathtt{n}\ +\ \mathtt{2})
\end{array}
$$

Figure 3: A simple example of a task parallel program.

$\vec{r_\delta}$ are regions in which the new task can read the region variables, and $\vec{r_\omega}$ are regions in which the task can write region variables. Let `Stmts` be the set of all statements and let `Rets` $\subseteq$ (`Vals` $\rightarrow$ `Stmts`) be the set of return value handlers. The handler d associates the return value of the procedure with a user defined statement.

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. The rest of the tasks in the region r continue their execution normally and are joined when another **ewait** or **await** statement is issued by the parent task. A task is termed *completed* when its statement is a **return**-statement.

The **assume**-statement blocks a task until its expression e evaluates to **true**. By way of definition, **call**, **return**, **post**, **ewait** and **await** are *inter-procedural* statements. All other statements are *intra-procedural*.

Fig. 3 shows a simple example program. The main task posts two new tasks t_1 and t_2 executing procedures p_1 and p_2 in region r_1. ε denotes an empty region sequence. The tasks t_1 and t_2 have access to the variable r_1. The *main* task awaits the completion of t_1 and t_2. The return value handler of procedure *main* takes the value returned by the tasks t_1 and t_2 and updates the value of n. The computation graph for this program is that in Fig. 1.

The semantics is defined over trees of procedure frames to represent the parallelism in the language rather than stacks which are inherently sequential. That means that the frame of each posted task becomes a child to the parent's frame. The parent-child relationship is transferred appropriately with task passing or when a parent completes without synchronizing with its children. The evolution of the program proceeds by a task either taking an intra-procedural step, posting a new child frame, or removing a frame for a synchronized completed task.

A task $t = \langle \ell, s, d, \vec{r_\delta}, \vec{r_\omega}, n \rangle$ is a tuple containing the valuation of the procedure local variable `l`, along with a statement s, a return value handler d, a list of regions that it may use for read variables, a list of regions it may use for write variables, and an associated node in the computation graph for this task. When a procedure p

is posted as a task, the statement s is the statement defined for the procedure p—recall that statements are inductively defined.

A *tree configuration*, $c = \langle t, m \rangle$, is an inductively defined tree with task-labeled vertexes, t, and region labeled edges given by the *region valuation* function, $m : \texttt{Regs} \to \mathbb{M}[\texttt{Configs}]$, where $\texttt{Configs}$ is the set of tree configurations and $\mathbb{M}[\texttt{Configs}]$ are configuration multi-sets. For a given vertex $c = \langle t, m \rangle$, $m(r)$ returns the collection of sub-trees connected to the t-labeled root by r-labeled edges.

The semantics relies on manipulating region valuations for task passing between parents and children. For two region valuations m_1 and m_2, the notation $m_1 \cup m_2$ is the multi-set union of each valuation. Further, the notation $m \mid_{\vec{r}}$ is the projection of m to the sequence $\vec{r}$ defined as $m \mid_{\vec{r}}(r') = m(r')$ when r' is found somewhere in $\vec{r}$, and $m \mid_{\vec{r}}(r') = \emptyset$ otherwise.

Let $[\![\cdot]\!]_e$ be an evaluation function for expressions without any program or region variables such that $[\![\star]\!]_e = \texttt{Vals}$, and let $\ell(r)$ denote the value of the region variable in r. For convenience in the semantics definition, an evaluation function is defined over a task t that enforces the read rights assigned to the task:

$$\begin{aligned} e(t) &= e(\langle \ell, s, d, \vec{r_\delta}, \vec{r_\omega}, n \rangle) \\ &= e(\ell, \vec{r_\delta}) \\ &= e(\ell, r_0, r_1, \ldots) \\ &= [\![e[\ell/\mathtt{l}, \ell(r_0)/\mathtt{l}(r_0), \ell(r_1)/\mathtt{l}(r_1), \ldots]]\!]_e \end{aligned}$$

If $e[\ell/\mathtt{l}, \ell(r_0)/\mathtt{l}(r_0), \ell(r_1)/\mathtt{l}(r_1), \ldots]$ has any free variables, then by definition,
$[\![e[\ell/\mathtt{l}, \ell(r_0)/\mathtt{l}(r_0), \ell(r_1)/\mathtt{l}(r_1), \ldots]]\!]_e$ has no meaning and is undefined (i.e., $e(t) = \emptyset$). e(t) is set-valued, although most expressions evaluate to singletons. As a final convenience for dealing with expressions in the semantics when constructing computation graphs, let the set of regions whose variables appear in e be denoted by $\eta(e)$.

Contexts are used to further simplify the notation needed to define the semantics. A *configuration context*, C, is a tree with task-labeled vertexes, region-labeled edges and a single $\diamond$-labeled leaf. The notation $C[c]$ denotes the configuration obtained by substituting a configuration c for the unique $\diamond$-labeled leaf of C. The configuration isolates individual task transitions (e.g., $C[\langle t, m \rangle] \to C[\langle t', m \rangle]$ denotes an intra-procedural transition on a task). Similarly, a *statement context* is given as $S = \diamond; s_1; \ldots; s_i$ and $S[s]$ indicates that $\diamond$ is replaced by s where s is the next statement to be executed. A *task statement context*, $T = \langle \ell, S, d, \vec{r_\delta}, \vec{r_\omega}, n \rangle$ is a task with a statement context in place of a statement, and likewise $T[s]$ indicates that s is the next statement to be executed in the task. Like configuration contexts, task statement contexts isolate the statement to be executed (e.g., $C[\langle T[s_1], m \rangle] \to C[\langle T[s_2], m \rangle]$ denotes an intra-procedural transition that modifies the statement in some way). For convenience, $e(t)$ is naturally extended to use contexts as indicated by $e(T)$.

As indicated previously, a task t is completed when its next to be executed statement s is **return** e. The set of possible return-value handler statements for t is $\mathrm{rvh}(t) = \{d(\ell) \mid \ell \in e(T)\}$ given the task's context. By defnition, $\mathrm{rvh}(t) = \emptyset$ when t is not completed or $e(T)$ is undefined.

Call

$$\frac{}{\begin{array}{l} C[T[\textbf{call}\ \mathtt{l}\ := p\ e\ \vec{r_\delta}\ \vec{r_\omega}], m] \to \\ C[T[\textbf{post}\ r_{call} \leftarrow p\ e\ \varepsilon\ \vec{r_\delta}\ \vec{r_\omega}\ \lambda v.\mathtt{l} := v;\ \textbf{ewait}\ r_{call}], m] \end{array}}$$

Post

$$\frac{\begin{array}{ll} n_0' = \mathrm{fresh}() & n_1 = \mathrm{fresh}() \\ N = N \cup \{n_0', n_1\} & E = E \cup \{\langle n_0, n_0' \rangle, \langle n_0, n_1 \rangle\} \\ \ell \in e(\ell', \vec{r_\delta}') & \delta = \delta \cup (n_0 \mapsto \eta(e)) \\ \multicolumn{2}{l}{m' = (m \setminus m|_{\vec{r}}) \cup (r \mapsto \langle \langle \ell, s_p, d, \vec{r_\delta}, \vec{r_\omega}, n_1 \rangle, m|_{\vec{r}} \rangle)} \end{array}}{\begin{array}{l} C[\langle \ell', S[\textbf{post}\ r \leftarrow p\ e\ \vec{r}\ \vec{r_\delta}\ \vec{r_\omega}\ d], \vec{r_\delta}', \vec{r_\omega}', d', n_0 \rangle, m] \to \\ C[\langle \ell', S[\textbf{skip}], \vec{r_\delta}', \vec{r_\omega}', d', n_0' \rangle, m'] \end{array}}$$

Ewait

$$\frac{\begin{array}{lll} n' = \mathrm{fresh}() & N = N \cup \{n'\} & E = E \cup \{\langle n, n' \rangle, \langle n(t_2), n' \rangle\} \\ m_1 = (r \mapsto \langle t_2, m_2 \rangle) \cup m_1' & s \in \mathrm{rvh}(t_2) & \end{array}}{\begin{array}{l} C[\langle \ell, S[\textbf{ewait}\ r], \vec{r_\delta}, \vec{r_\omega}, d, n \rangle, m_1] \to \\ C[\langle \ell, S[s], \vec{r_\delta}, \vec{r_\omega}, d, n' \rangle, m_1' \cup m_2] \end{array}}$$

Await-next

$$\frac{\begin{array}{lll} n' = \mathrm{fresh}() & N = N \cup \{n'\} & E = E \cup \{\langle n, n' \rangle, \langle n(t_2), n' \rangle\} \\ m_1 = (r \mapsto \langle t_2, m_2 \rangle) \cup m_1' & s \in \mathrm{rvh}(t_2) & \end{array}}{\begin{array}{l} C[\langle \ell, S[\textbf{await}\ r], \vec{r_\delta}, \vec{r_\omega}, d, n \rangle, m_1] \to \\ C[\langle \ell, S[s;\ \textbf{await}\ r], \vec{r_\delta}, \vec{r_\omega}, d, n' \rangle, m_1' \cup m_2] \end{array}}$$

Await-done

$$\frac{m(r) = \emptyset}{C[T_1[\textbf{await}\ r], m] \to C[T_1[\textbf{skip}], m]}$$

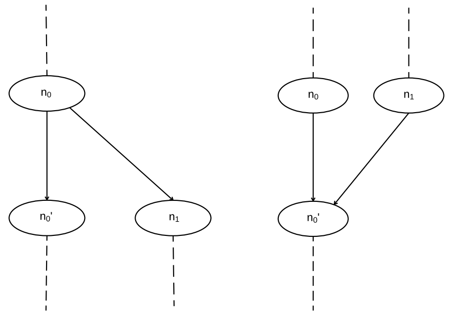
Figure 4: The transition rules for the inter-procedural statements.

The initial condition for a program $\iota = \langle p, \ell \rangle$ is an initial procedure $p \in \texttt{Procs}$ and an initial value $\ell \in \texttt{Vals}$. The initial configuration is created from ι as $c = \langle \langle \ell, s_p, d, \vec{r_\delta}, \vec{r_\omega}, n \rangle, m \rangle$, where s_p is the statement for the procedure p, d is the identity function (i.e., $\lambda v.v$), $\vec{r_\delta}$ list regions whose variables are read by p, $\vec{r_\omega}$ lists regions whose variables are written by p, n is a fresh node for the computation graph (i.e., $n = \mathrm{fresh}()$), and $\forall r \in \texttt{Regs}, m(r) = \emptyset$.

The semantics is now given as a set of transition rules relating tree configurations. The rules assume the presence of a global computation graph, $G = \langle N, E, \delta, \omega \rangle$, that is updated as part of the transition. The initial graph contains a single node $N = \{n\}$ from the initial configuration, no edges ($E = \emptyset$), and no read/write information ($\delta(n) = \emptyset$ and $\omega(n) = \emptyset$).

The intra-procedural transition rules are omitted for space but are defined in the usual way. Fig. 4 shows semantics for the inter-procedural statements. The notation, $\delta = \delta \cup (n \mapsto \eta(e))$, is understood to update δ such that n additionally maps to $\eta(e)$. A function notation is adopted to access the tuple $T = \langle \ell, S, d, \vec{r_\delta}, \vec{r_\omega}, n \rangle$. For example, $\vec{r_\omega}(T)$ indicates the read-region vector in the task or task context.

The **call** statement is interpreted as a **post** followed by **ewait** on some region r_{call}. This region r_{call} is exclusive to the task calling the procedure and cannot be used to post new tasks into this region. A call statement does not allow ownership of any tasks



(a) Task creation. (b) Task synchronization.

Figure 5: Steps involved in computation graph creation.

to be passed to the newly created task. The region variables that are available to this task for reading and writing are denoted by $\vec{r_\delta}$ and $\vec{r_\omega}$ respectively.

The Post rule is fired when the task forks to create a new child task that potentially runs in parallel with the parent task. When a task t_1 executes a **post** statement, two fresh nodes n_0' and n_1 are added to the graph. Node n_0' represents the statements following **post** and n_1 represents the statements executed by t_2. The current node n_0 of t_1 is connected to n_0' and n_1 as shown in Fig. 5(a). The read set δ of node n_0 is updated to additionally map to the regions in $\eta(e)$ (i.e., the regions referenced in the expression e). The current node of t_1 changes to n_0' after the transition. The region mapping m of task t_1 is updated by removing the configurations of regions whose ownership is passed to the newly created task t_2 and adding a new configuration that consists of the task t_2 along with the regions it now owns.

The Ewait rule blocks the execution of the currently executing task until a task in the indicated region completes. The choice of completed task, t_2, in the region is non-deterministic. A node n' is added to the graph to act as a join node. It captures the subsequent statements of t_1 after the **ewait** statement finishes. The current node n of task t_1 and the current node of task t_2, denoted by $n(t_2)$ are connected to n' as shown in Fig. 5(b). The configuration $(r \mapsto \langle t_2, m_2 \rangle)$ is removed from the region valuation m of t_1. After the transition, the current node of task t_1 is changed to n'. The task t_1 is resumed with a return value handler for the completed task ($\text{rvh}(t_2)$) before continuing with its next statement.

The Await-Next rule blocks the execution of the currently executing task t_1 until all the tasks owned by t_1 whose handles are stored in region r complete execution. The rule is implemented recursively by removing one task from the region at a time and then inserting another **await**-statement on the same region. A join node n' is added to the graph, the current nodes of t_1 and t_2 are connected to n' as shown in Fig. 5(b) and the current node of t_1 is changed to n'. When task t_2 returns a value to t_1, t_1 executes the statement from the return value handler $\text{rvh}(t_2)$. The Await-Done rule terminates recursion when the region is empty.

The computation graph for the example in Fig. 3 is presented in Fig. 1. The order of synchronization of tasks t_1 and t_2 affects the value of the variable r_1 in the *main* task. The return value handlers of the tasks get executed in different orders under different program schedules. This makes the output of the program non-deterministic. In a schedule where task t_1 joins *main* task before t_2, the value of n at the end of program execution is 3 and in a schedule where task t_2 joins *main* task before t_1, the value of r_1 is 2.

Theorem 3.1. *Using the tree semantics with Algorithm 1 to detect data-race in the resulting computation graph is complete for a task parallel program with a given input.*

4 A DETERMINISTIC FRAGMENT OF THE MODEL

This section defines a deterministic fragment of the task parallel language such that a computation graph from a single execution is sufficient to prove or disprove data-race. In other words, data-race detection using Algorithm 1 on the computation graph from an single execution of a program in this fragment is both sound and complete meaning that it neither under-approximates (sound) nor over-approximates (complete) the set of data-race free programs. The fragment is defined by the following language restrictions:

- Passing ownership of tasks from a parent to a child task is not allowed.
- Tasks whose return value handlers side effect can be posted in single-task regions only (i.e., regions that contain only a single task). A side-effect of a return value handler can be a change in the state of either the local variable or a region variable.
- All the tasks are joined to the main task at the end of the program execution. This is ensured by having the initial program configuration as $\langle \text{T}[\textbf{post}\ r_0 \leftarrow p_0\ e\ \varepsilon\ \vec{r}\ \vec{r}\ \lambda \text{v.v};\ \textbf{await}\ r_0; \textbf{await}\ r_1; \ldots], m_0\rangle$ on some procedure p_0, $\vec{r}$ is the region sequence containing all regions and $\forall r \in \texttt{Regs}$, $m_0(r) = \emptyset$

Fig. 6 is an example, with a data-race, from the equivalent fragment in the Habanero model to show the relationship to real-world programming languages.

In Habanero, the **async** construct creates a new asynchronous task that runs in parallel with the parent task. The **finish** construct is used to collectively synchronize children tasks with their parent task. The **finish** s statement causes the parent task to execute s and then wait until all tasks created inside the finish-block have completed. The future construct lets tasks return values to other tasks with the operation `f.get()` that blocks until the task associated with f completes.

Fig. 7 is the equivalent program in the model in this presentation. The procedure *main* posts task from the outer finish block to region r_1 and tasks from the inner finish block to region r_2. Since, the inner finish block completes execution first, **await** on region r_2 is called before r_1. The future posts a task to region r_3 followed by an **ewait** on r_3.

Let $\mathcal{G}(P)$ return the set of computation graphs from all possible schedules of the program P from the deterministic fragment of the model, and let DRF(G) return true if Algorithm 1 reports the graph to be data race free.

Lemma 4.1. $(\exists G \in \mathcal{G}(P),\ \text{DRF}(G)) \rightarrow (\forall G \in \mathcal{G}(P),\ \text{DRF}(G))$

```
public class Example1{
    static int x = 0;
    public static void main(String [] args) {
        finish {
            async { //Task1
                x = x + 1;
            }
            finish{
                async { //Task2
                    x = x + 2;
                }
            }
        }
        future f = async { //Task3
                            return 5;
                         }
        x = f.get();
    }
}
```

Figure 6: An example of a Habanero Java Program.

```
proc $main$ (var n : int)
  $\mathtt{l}(r_1) := 0$;
  post $r_1 \leftarrow p_1\ 0\ \varepsilon\ \vec{r}\ \vec{r}\ \lambda n.n$;
  post $r_2 \leftarrow p_2\ 0\ \varepsilon\ \vec{r}\ \vec{r}\ \lambda n.n$;
  await $r_2$;
  await $r_1$;
  post $r_3 \leftarrow p_3\ 0\ \varepsilon\ \vec{r}\ \vec{r}\ \lambda n.\mathtt{l}(r_1) := 5$;
  ewait $r_3$;
proc $p_1$ (var n : int)
  $\mathtt{l}(r_1) := \mathtt{l}(r_1) + 1$
proc $p_2$ (var n : int)
  $\mathtt{l}(r_1) := \mathtt{l}(r_1) + 2$
proc $p_3$ (var n : int)
  return 5
```

Figure 7: Converted version of the Habanero Java program from Fig. 6.

The proof is omitted for space (as are the other non-trivial proofs), but the lemma states that if data-race is not detected in an observed execution of a program from the deterministic fragment, then all other possible executions are data-race free as well; in other words, programs from the fragment are deterministic. Habanero makes this same claim but does not prove it [9]. The corollary regarding data-race programs and the following theorem are trivial from Lemma 4.1.

Corollary 4.2. $(\exists G \in \mathcal{G}(P),\ \neg\mathrm{DRF}(G)) \rightarrow (\forall G \in \mathcal{G}(P),\ \mathrm{DRF}(G))$

Theorem 4.3. *Using the tree semantics with Algorithm 1 to detect data-race in the resulting computation graph is sound and complete for a task parallel program with a given input when that program is in the deterministic fragment of the language.*

5 MODEL CHECKING

Programs that use mutual exclusion to protect accesses to shared variables introduce non-determinism as different access orders in the critical sections may lead to different computation graphs. Model checking can be used to enumerate each of these computation graphs. The task parallel language is extended to model mutual exclusion with a new statement: **isolated** s. The statement performs s in mutual exclusion of any other isolated statements. The semantics with the computation graph construction is in Fig. 8. The isolation is accomplished by creating a new global variable *last* to track the last node in the computation graph belonging to an isolated statement, by adding to the task context a counter initialized to zero to count the number of nested isolated contexts, and with a new keyword for the rewrite rules: **isolated-end**.

Let canIsolate(C) be a function over configurations to Boolean that returns true for a configuration tree if all the task counters are 0; otherwise it returns false. If no other isolated statements are running, then the Isolated rule increments the task counter to indicate isolation and inserts after the isolated statement s the new **isolated-end** keyword. 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 note, *last* is initialized to an empty node when execution starts. The Isolated-Nested rule simply increments the counter if the task is already in isolation.

The Isolated-End-Nested rule processes the new **isolated-end** keyword and decrements the counter. When the counter reaches the outer-most isolated context, the Isolated-End rule creates a new node in the computation graph to denote the end of isolation, and it updates *last* to properly sequence any future isolation. As a note, the on-the-fly data race detection is modified to not reduce sub-graphs with isolation.

Algorithm 2 presents a scheduling algorithm to enumerate all computation graphs resulting from isolation for model checking [36]. The algorithm considers a simplified state of the program with `Regs` being the set of region variables that are shared among the tasks, `Tasks` being the set of tasks, t being a task, and R being the set of runnable tasks. The algorithm also implements *sequential semantics*, or depth-first semantics, where only one task runs at a time and that task runs until it waits, completes, or isolates at which time a scheduling choice is made. Sequential semantics are viable by Lemma 4.1 and Corollary 4.2 that establish independence in the computation graph and execution schedule in the absence of data races.

Line 2 updates the region variables and pool of tasks by running task t until it exits, waits, or reaches an **isolated**-construct. The function `status` on Line 3 returns the status of the task t. On Line 4, the function `runnable` is used to obtain a list of all the tasks that can be run from the pool of all tasks. If the status of the currently running task t becomes ISOLATED (i.e., the task encounters an **isolated** construct), the task is preempted and all the tasks that are runnable, including the task that is trying to isolate,

Isolated

$$\frac{\text{canIsolate(C)} = true \qquad n' = \text{fresh}() \qquad N = N \cup \{n'\} \qquad E = E \cup \{\langle n, n'\rangle, \langle last, n'\rangle\}}{C[\langle \ell', S[\textbf{isolated } s], \vec{r_\delta}', \vec{r_\omega}', d', n, 0\rangle, m] \to C[\langle \ell', S[s; \textbf{isolated-end}], \vec{r_\delta}', \vec{r_\omega}', d', n', 1\rangle, m]}$$

Isolated-Nested

$$\frac{iso > 0 \qquad iso' = iso + 1}{C[\langle \ell', S[\textbf{isolated } s], \vec{r_\delta}', \vec{r_\omega}', d', n, iso\rangle, m] \to C[\langle \ell', S[s; \textbf{isolated-end}], \vec{r_\delta}', \vec{r_\omega}', d', n, iso'\rangle, m]}$$

Isolated-End-Nested

$$\frac{iso > 1 \qquad iso' = iso - 1}{C[\langle \ell', S[\textbf{isolated-end}], \vec{r_\delta}', \vec{r_\omega}', d', n, iso\rangle, m] \to C[\langle \ell', S[\textbf{skip}], \vec{r_\delta}', \vec{r_\omega}', d', n, iso'\rangle, m]}$$

Isolated-End

$$\frac{n' = \text{fresh}() \qquad last = n \qquad N = N \cup \{n'\} \qquad E = E \cup \{\langle n, n'\rangle\}}{C[\langle \ell', S[\textbf{isolated-end}], \vec{r_\delta}', \vec{r_\omega}', d', n, 1\rangle, m] \to C[\langle \ell', S[\textbf{skip}], \vec{r_\delta}', \vec{r_\omega}', d', n', 0\rangle, m]}$$

Figure 8: The transition rules for isolated statements.

Algorithm 2 Scheduling algorithm for Isolated blocks

```
1:  function SCHEDULE($t$, Regs, Tasks)
2:      loop: (Regs, Tasks) := run($t$, Regs, Tasks)
3:      $s$ := status($t$)
4:      $R$ := runnable(Tasks)
5:      if $s$ = ISOLATED then
6:          for all $t_i \in R$ do
7:              schedule($t_i$, Regs, Tasks)
8:          end for
9:      else
10:         $t_i$ := random($R$)
11:         schedule($t_i$, Regs, Tasks)
12:     end if
13: end function
```

are scheduled by the runtime meaning that the model checker considers all ways to interleave the runnable tasks at that point. When the task completes, a new task is randomly selected from the set of runnable tasks.

Theorem 5.1. *Algorithm 2 creates all possible computation graphs due isolation.*

Model checking enumerates and checks each possible computation graph from the program to determine if it is data-race free. Since every graph is considered, and the technique is sound and complete, only programs that are truly data-race free are accepted.

6 IMPLEMENTATION AND RESULTS

The data race detection technique described in this paper has been implemented for Habanero Java. It uses the verification runtime specifically designed to test HJ programs [2]. The runtime is written in a way to enable JPF to fully verify HJ programs without any modification to JPF. JPF is given a new *PropertyListenerAdapter* to create the computation graph, and it is given a new scheduling-factory based on Algorithm 2 that implements sequential semantics as defined in Section 5 and enumerates all the computation graphs arising from isolation. The resulting computation graphs are analyzed for data-race using Algorithm 1.

The results from the JPF implementation have been compared to two other approaches implemented by JPF: *Precise race detector* (PRD) and *Gradual permission regions* (GPR) on benchmarks that cover a wide range of functionality in HJ. They spawn a wide range of tasks with smaller programs having 3-15 tasks going all the way up to 525 tasks for larger programs. The experiments were run on a machine with an Intel Core i5 processor with 2.6GHz speed and 8GB of RAM. The results show a significant improvement in the time required for verification (see Table 1).

In Table 1, the number of states explored by JPF and time required for verification by each method is compared. The tests are run for a maximum of an hour before they are terminated manually. If a test does not finish in the time bound or if it runs out of JVM memory, then it is marked as N/A in the table. The error note column shows the results of verification. The tests that produce erroneous results are marked with an asterisk (∗). The results are not averaged over several runs to account for *luck* in the search order.

The PRD algorithm is a partial order reduction based on JPF's ability to detect shared memory accesses (e.g., it tracks thread IDs on all heap accesses). With PRD, JPF only considers schedules around shared memory accesses. This partial order is JPF's default search. The PRD algorithm merely flags an error when it sees a conflicting access at a memory location by two different threads. PRD generally does not complete execution within the stipulated time or runs out of memory even on smaller programs because of the state space explosion. It also reports race for *Two Dimensional Arrays*, *Scalar multiply* and *Vector Add* benchmarks where no data race actually exists in the program. This error is because in PRD, the access on an array object looks like a data race since it is not able to see the difference in the indexes—a shortcoming in the PRD implementation.

GPR uses program annotations to reduce the number of shared locations that need to consider scheduling [36]. GPR works better than PRD because GPR groups several bytecodes that access shared

Table 1: Benchmarks of HJ programs: Computation graphs vs Permission Regions vs. PreciseRaceDetector

			Computation graphs			*Gradual permission regions*			*Precise race detector*		
Test ID	**SLOC**	**Tasks**	**States**	**Time**	**Error Note**	**States**	**Time**	**Error Note**	**States**	**Time**	**Error Note**
Primitive Array Race	39	3	5	00:00	Race	5	00:00	Race	220	00:00	Race
Substring Search	83	59	64	00:03	Race	8	00:00	Race	N/A	N/A	N/A
Reciprocal Array Sum	58	2	4	00:08	Race	32	00:06	Race	N/A	N/A	N/A
Primitive Array No Race	29	3	5	00:00	No Race	5	00:00	No Race	11,852	00:00	No Race
Two Dim Arrays	30	11	15	00:00	No Race	15	00:00	No Race	597	00:00	Race*
ForAll With Iterable	38	2	9	00:00	No Race	9	00:00	No Race	N/A	N/A	N/A
Integer Counter Isolated	54	10	24	00:01	No Race	1,013,102	05:53	No Race	N/A	N/A	N/A
Pipeline With Futures	69	5	34	00:00	No Race	34	00:00	No Race	N/A	N/A	N/A
Binary Trees	80	525	630	00:25	No Race	632	00:03	No Race	N/A	N/A	N/A
Prime Num Counter	51	25	776	00:01	No Race	3,542,569	17:37	No Race	N/A	N/A	N/A
Prime Num Counter ForAll	52	25	30	00:02	No Race	18	00:01	No Race	N/A	N/A	N/A
Prime Num Counter ForAsync	44	11	653	00:01	No Race	2,528,064	15:44	No Race	N/A	N/A	N/A
Add	67	3	11	00:01	No Race	62,374	00:33	No Race	4930	00:03	Race*
Scalar Multiply	55	3	15	00:01	No Race	55,712	00:30	No Race	826	00:01	Race*
Vector Add	50	3	5	00:00	No Race	17	00:00	No Race	46,394	00:19	No Race
Clumped Access	30	3	5	00:03	No Race	15	00:00	No Race	N/A	N/A	N/A

locations into a single atomic block of code with read/write indications. For example, if there are two bytecodes that touch shared memory locations, PRD schedules from each of the two locations. In contract, 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 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. As such regardless of how the annotations are indicated (automatic or manual), the approach has to consider schedules at each annotation, which leads quickly to state explosion.

GPR falls behind quickly as the number of regions grow compared to the computation graph solution. The difference in performance is seen in the *Add*, *Scalar multiply* and *Prime number counter* benchmarks which used shared variables that lead to several regions (which are as big as possible without creating a data-race–another issue with GPR in general). The *Prime number counter* benchmark also has isolated sections and therefore, 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.

We also evaluated our data race detector on some real world benchmarks. The *Crypt-af* and *Crypt-f* benchmarks are implementation of the IDEA encrytion 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 [49]. Table 2 shows the results of this evaluation. These are quickly verified free of data-race using computation graphs; whereas, the other two approaches time out.

Table 2: Evaluation of Computation graphs on real world benchmarks

Test ID	**SLOC**	**Tasks**	**States**	**Time**	**Error Note**
Crypt-af	1010	259	260	00:17	No Race
Crypt-f	1145	387	775	00:46	No Race
Series-af	730	329	750	00:36	No Race
Series-f	830	354	630	00:51	No Race
Strassen	560	3	7	00:57	No Race

7 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 [13, 29, 31, 42, 48] and then improves precision with dynamic analysis [12, 15, 20, 23, 32]. Other approaches reason about threads individually [21, 24, 25, 34, 53], rely on assertions [7, 8, 26, 28, 33, 46, 54, 55], use low-overhead instrumentation [38], or construct type proofs [1]. 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 [16–18, 30, 37, 41, 45, 50]. Locking protocols are not directly applicable to task-parallel programming models that also constrain parallelism but often without explicit locking. It should be said though that static analysis of structured programs can be efficient in detecting data-race but typically not in a way that is both sound and complete as any static analysis will depend on how shared memory locations are identified. Typically, such analyses over-approximate 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 [3, 11, 19, 40, 47]. The method has also been applied to the Habanero programming model to support a limited set of Habanero keywords including futures but not isolation [43, 44]. Compared to the solution for data-race detection in this paper, the SP-bags algorithm relies on the underlying computation graph, but rather than construct the graph explicitly, it implicitly uses the graph to track parallel memory references. This implicit graph representation leads to a very efficient implementation based on disjoint sets that can be integrated into the runtimes in a way that mitigates the overhead of data-race detection. Although the slowdown is significant (an order of magnitude in general), it is tolerable enough to run very large problem instances. That said, these approaches are dynamic. So they may claim a program data-race free when in fact it is not. Also, they do not have the ability to enumerate all possible executions even in the presence of isolation. Adding new parallelism to these approaches is also non-trivial as shown with the addition of the future keyword in the Habanero model [43, 44]. Building the graph directly makes adding keywords more direct. The trade-off though is that the solution in this paper is less efficient, but it does not need to be super efficient. It never intends to scale to thousands of tasks (JPF is not able to do that) nor is it intended to run on massive problem instances; rather, it builds on the fact that model checking relies critically a small problem instances, and that small instances are sufficient to capture all the interesting task interactions. The goal in the approach in this paper is verification and not run time monitoring, so the simpler less efficient approach is both acceptable and preferred for arguing the approach is correct.

Programmer annotations indicating shared interactions (e.g., permission regions) do improve model checking in general [51, 52]. 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. A partial order reduction algorithm will enumerate all schedules around a shared variable looking for a data-race whereas the approach in this paper analyzes a single trace and concludes if a data-race exists or not. It critically matters when a program is data-race free. In the absence of isolation, the approach in this paper is done after a single execution whereas the partial order reduction is enumerating schedules. In the presence of isolation, the difference between the two is less obvious depending on the number of isolated statements, but it is expected that the approach in this paper will degrade similarly to a partial order reduction meaning that as the number of isolated statements increase so do the number of unique computation graphs that must be considered.

There are other model checkers for task parallel languages [22, 56]. The first modifies JPF and an X10 runtime 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 runtime 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. The key is to start with a runtime intended for verification and not high performance. From there, with a tool such as JPF, it is easy to tweak JPF's default behavior to schedule as needed and analyze the computation graphs.

8 CONCLUSION

This work presents a sound and complete technique for data race detection in task parallel programs using computation graphs. The computation graph creation is presented with the formal semantics for task parallel languages. A scheduling algorithm to create all computation graph structures for programs containing mutual exclusion is also presented for use in model checking. 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.

REFERENCES

[1] Martin Abadi, Cormac Flanagan, and Stephen N Freund. 2006. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 2 (2006), 207–255.

[2] Peter Anderson, Brandon Chase, and Eric Mercer. 2014. JPF verification of Habanero Java programs. *ACM SIGSOFT Software Engineering Notes* 39, 1 (2014), 1–7.

[3] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-fly Maintenance of Series-parallel Relationships in Fork-join Multithreaded Programs. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*. ACM, New York, NY, USA, Article 1, 12 pages. DOI: http://dx.doi.org/10.1145/1007912.1007933

[4] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69.

[5] Ahmed Bouajjani and Michael Emmi. 2012. Analysis of recursively parallel programs. *ACM SIGPLAN Notices* 47, 1 (2012), 203–214.

[6] J Mark Bull, Lorna A Smith, Martin D Westhead, David S Henty, and Robert A Davey. 2000. A benchmark suite for high performance Java. *Concurrency - Practice and Experience* 12, 6 (2000), 375–388.

[7] Jacob Burnim and Koushik Sen. 2009. Asserting and checking determinism for multithreaded programs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, ACM, ACM, 3–12.

[8] Jacob Burnim and Koushik Sen. 2010. DETERMIN: Inferring likely deterministic specifications of multithreaded programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, ACM, ACM, 415–424.

[9] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. ACM, ACM, ACM, 51–61.

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.* 40, 10, Article 1 (Oct. 2005), 20 pages. DOI: http://dx.doi.org/10.1145/1103845.1094852

[11] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 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, Article 1, 12 pages. DOI: http://dx.doi.org/10.1145/277651.277696

[12] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. *SIGPLAN Not.* 37, 5, Article 1 (May 2002), 12 pages. DOI:http://dx.doi.org/10.1145/543552.512560
[13] Jong-Deok Choi, Alexey Loginov, and Vivek Sarkar. 2001. *Static datarace analysis for multithreaded object-oriented programs.* Technical Report. Technical Report RC22146, IBM Research.
[14] Jack B Dennis, Guang R Gao, and Vivek Sarkar. 2012. Determinacy and repeatability of parallel program schemata. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012.* IEEE, IEEE, IEEE, 1–9.
[15] Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. 2014. Commutativity race detection. In *ACM SIGPLAN Notices*, Vol. 49:6. ACM, ACM, ACM, 305–315.
[16] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2006. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Formal Approaches to Software Testing and Runtime Verification.* Springer, Springer, 193–208.
[17] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: a race and transaction-aware Java runtime. In *ACM SIGPLAN Notices*, Vol. 42:6. ACM, ACM, ACM, 245–255.
[18] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, Vol. 37:5. ACM, ACM, ACM, 237–252.
[19] Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97).* ACM, New York, NY, USA, Article 1, 11 pages. DOI:http://dx.doi.org/10.1145/258492.258493
[20] Cormac Flanagan and Stephen N Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, Vol. 44:6. ACM, ACM, ACM, 121–133.
[21] Cormac Flanagan and Shaz Qadeer. 2003. Thread-modular model checking. In *Model Checking Software.* Springer, Springer, 213–224.
[22] Milos Gligoric, Peter C Mehlitz, and Darko Marinov. 2012. 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, 11–20.
[23] Patrice Godefroid. 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, Article 1, 13 pages. DOI:http://dx.doi.org/10.1145/263699.263717
[24] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. 2007. Thread-modular shape analysis. In *ACM SIGPLAN Notices*, Vol. 42:6. ACM, ACM, ACM, 266–277.
[25] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. 2003. *Thread-Modular Abstraction Refinement.* Springer Berlin Heidelberg, Berlin, Heidelberg, 262–274. DOI:http://dx.doi.org/10.1007/978-3-540-45069-6_27
[26] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. 2012. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis.* ACM, ACM, ACM, 210–220.
[27] Shams Imam and Vivek Sarkar. 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, ACM, ACM, 75–86.
[28] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. 2015. Unfolding based automated testing of multithreaded programs. *Automated Software Engineering* 22, 4 (2015), 475–515.
[29] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. 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 ACM SIGSOFT symposium on The foundations of software engineering.* ACM, ACM, ACM, 13–22.
[30] Vineet Kahlon and Chao Wang. 2010. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Computer Aided Verification.* Springer, Springer, Springer, 434–449.
[31] Sergey Kulikov, Nastaran Shafiei, Franck Van Breugel, and Willem Visser. 2010. Detecting Data Races with Java PathFinder. (2010).
[32] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
[33] Hernan Ponce de Leon, Olli Saarikivi, Kari Kahkonen, Keijo Heljanko, and Javier Esparza. 2015. Unfolding Based Minimal Test Suites for Testing Multithreaded Programs. In *Application of Concurrency to System Design (ACSD), 2015 15th International Conference on.* IEEE, IEEE, IEEE, 40–49.
[34] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. 2007. Precise thread-modular verification. In *Static Analysis.* Springer, Springer, 218–232.
[35] John Mellor-Crummey. 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, Article 1, 10 pages. DOI:http://dx.doi.org/10.1145/125826.125861
[36] Eric Mercer, Peter Anderson, Nick Vrvilo, and Vivek Sarkar. 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, ACM, ACM, 535–540.
[37] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. *SIGPLAN Not.* 41, 6, Article 1 (June 2006), 12 pages. DOI:http://dx.doi.org/10.1145/1133255.1134018
[38] Adrian Nistor, Darko Marinov, and Josep Torrellas. 2010. Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, IEEE, IEEE, 251–262.
[39] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise dynamic datarace detection for structured parallelism. In *ACM SIGPLAN Notices*, Vol. 47:6. ACM, ACM, ACM, 531–542.
[40] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin T. Vechev, and Eran Yahav. 2012. Efficient data race detection for async-finish parallelism. *Formal Methods in System Design* 41, 3 (2012), 321–347.
[41] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.
[42] D Schonberg. 1989. *On-the-fly detection of access anomalies.* Vol. 24:7. ACM, ACM.
[43] Rishi Surendran and Vivek Sarkar. 2016. Dynamic Determinacy Race Detection for Task Parallelism with Futures. In *Runtime Verification, 2016 16th International Conference on.* Springer, Sprinter, Springer, Article 1, 2 pages.
[44] Rishi Surendran and Vivek Sarkar. 2016. *Dynamic Determinacy Race Detection for Task Parallelism with Futures.* Springer International Publishing, Cham, 368–385. DOI:http://dx.doi.org/10.1007/978-3-319-46982-9_23
[45] Serdar Tasiran Tayfun Elmas, Shaz Qadeer. 2005. *Precise race detection and efficient model checking using locksets.* Technical Report. MSR. https://www.microsoft.com/en-us/research/publication/precise-race-detection-and-efficient-model-checking-using-locksets/
[46] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. 2015. Recontest: Effective regression testing of concurrent programs. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, IEEE, IEEE, 246–256.
[47] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16).* ACM, New York, NY, USA, Article 1, 12 pages. DOI:http://dx.doi.org/10.1145/2935764.2935801
[48] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. 2011. Automatic verification of determinism for structured parallel programs. In *Static Analysis.* Springer, Springer, 455–471.
[49] Philippe Virouleau, Pierrick BRUNET, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. 2014. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *10th International Workshop on OpenMP, IWOMP2014 (10th International Workshop on OpenMP, IWOMP2014).* Springer, Salvador, Brazil, France, 16 – 29.
[50] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. 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, 205–214.
[51] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. 2012. Permission regions for race-free parallelism. In *Runtime Verification.* Springer, Springer, Springer, 94–109.
[52] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. 2012. Practical permissions for race-free parallelism. In *ECOOP 2012–Object-Oriented Programming.* Springer, Springer, 614–639.
[53] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. 1997. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing* 9, 2 (1997), 149–174.
[54] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: a coverage-driven testing tool for multithreaded programs. In *ACM Sigplan Notices*, Vol. 47:10. ACM, ACM, ACM, 485–502.
[55] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. 2014. SimRT: an automated framework to support regression testing for data races. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, ACM, ACM, 48–59.
[56] Timothy K Zirkel, Stephen F Siegel, and Timothy McClory. 2013. Automated Verification of Chapel Programs Using Model Checking and Symbolic Execution. *NASA Formal Methods* 7871 (2013), 198–212.