Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2016-10-01

Verification of Task Parallel Programs Using Predictive Analysis

Radha Vi Nakade
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation

Nakade, Radha Vi, "Verification of Task Parallel Programs Using Predictive Analysis" (2016). *All Theses and Dissertations*. 6176.
https://scholarsarchive.byu.edu/etd/6176

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

المنارة للاستشارات

www.manaraa.com

Verification of Task Parallel Programs Using Predictive Analysis

Radha Vi Jay Nakade

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Eric Mercer, Chair
Quinn Snell
Parris Egbert

Department of Computer Science

Brigham Young University

Copyright © 2016 Radha Vi Jay Nakade

All Rights Reserved

المنارة للاستشارات

www.manaraa.com

ABSTRACT

Verification of Task Parallel Programs Using Predictive Analysis

Radha Vi Jay Nakade
Department of Computer Science, BYU
Master of Science

Task parallel programming languages provide a way for creating asynchronous tasks that can run concurrently. The advantage of using task parallelism is that the programmer can write code that is independent of the underlying hardware. The runtime determines the number of processor cores that are available and the most efficient way to execute the tasks. When two or more concurrently executing tasks access a shared memory location and if at least one of the accesses is for writing, data race is observed in the program. Data races can introduce non-determinism in the program output making it important to have data race detection tools. To detect data races in task parallel programs, a new Sound and Complete technique based on computation graphs is presented in this work. The data race detection algorithm runs in $\mathcal{O}(N^2)$ time where N is number of nodes in the graph. A computation graph is a directed acyclic graph that represents the execution of the program. For detecting data races, the computation graph stores shared heap locations accessed by the tasks. An algorithm for creating computation graphs augmented with memory locations accessed by the tasks is also described here. This algorithm runs in $\mathcal{O}(N)$ time where N is the number of operations performed in the tasks. This work also presents an implementation of this technique for the Java implementation of the Habanero programming model. The results of this data race detector are compared to Java Pathfinder's precise race detector extension and permission regions based race detector extension. The results show a significant reduction in the time required for data race detection using this technique.

Keywords: verification, task parallel programming, data race detection

www.manaraa.com

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Eric Mercer, for his constant motivation and his patience. This work wouldn't have been possible without his help. I would also like to thank Dr. Jay McCarthy and my committee members for their feedback in the project.

I am grateful to my family for their love and support. And finally, I want to thank my wonderful husband, Atul, for his support, love and encouragement.

المنارة للاستشارات
www.manaraa.com

Table of Contents

List of Figures vi

List of Tables vii

1 Introduction 1

2 Data Race Detection 5

3 Computation Graphs 9

3.1 Surface Syntax 9

3.2 Tree-based Semantics 11

4 Structured Parallel Languages 22

4.1 Habanero Java 23

4.2 Properties of structured parallel programs 25

5 On-the-fly data race detection 28

6 Mutual Exclusion 33

7 Implementation and Results 39

7.1 Implementation 39

7.2 Results 40

8 Related Work 45

المنارة للاستشارات

www.manaraa.com

9 Conclusion and Future Work **48**

9.1 Conclusion and future work 48

References **49**

المنارة للاستشارات
www.manaraa.com

List of Figures

2.1 Computation Graph Example. 6

3.1 The surface syntax for task parallel programs. 9

3.2 A simple example of a task parallel program. 11

3.3 The transition rules for the intra-procedural statements. 14

3.4 The transition rules for the inter-procedural statements. 15

3.5 Steps involved in computation graph creation. 16

3.6 Parallel program with different computation graphs under different schedules. . . . 20

3.7 Computation graphs of example in Figure 3.6. 20

4.1 Example of an HJ Program. 23

4.2 HJ program converted to task parallel language. 24

5.1 AWAIT-DONE rule updated for on-the-fly data race detection. 30

5.2 A parallel program with nested regions. 30

5.3 Computation graph of example in Figure 5.2. 31

6.1 The transition rules for isolated statements. 34

6.2 Parallel program with mutual exclusion. 36

6.3 Computation graphs of example in Figure 6.2. 37

7.1 An HJ Program for comparing on-the-fly with normal data race detection using computation graphs. 44

المنارة للاستشارات

www.manaraa.com

List of Tables

7.1 Benchmarks of HJ programs: Computation graphs vs Permission Regions vs. PreciseRaceDetector. 41

7.2 Comparison of results for on-the-fly and normal data race detection using computation graphs. 43

المنارة للاستشارات

www.manaraa.com

Chapter 1

Introduction

The increasing use of multi-core processors is motivating the use of parallel programming. Earlier, the speed of processor cores was expected to increase rapidly with sustained technological advances and the need for parallel computing was relatively low. Now that processor speeds are no longer increasing, parallelism is the only way of obtaining higher computing performance.

Writing concurrent programs that are free from bugs, however, is very difficult because when programs execute different instructions simultaneously, different thread schedules and memory access patterns are observed that give rise to various issues such as data-races and deadlocks. Structured parallel languages help users to write parallel programs that are scalable and easy to maintain [1–3]. This flexibility is achieved by imposing restrictions on the way tasks can be forked and joined. The parallel constructs create regions where tasks are started and synchronized. This restriction ensures the parallel programs are deadlock free.

Data races occur in parallel programs when two or more tasks access a shared memory location such that at least one of the accesses is a write. A race on a shared variable can alter the value of the variable based on the order in which the variable is accessed by the tasks causing the output to be non-deterministic. A data race that is not protected (i.e., marked volatile) also leads to behavior that is not sequentially consistent. It is hard to test all possible outcomes of the program with a data race because the scheduler most often runs the tasks in the same order thereby producing the same result everytime. Data races might be benign, but they are generally an indication of a bug. Hence, it is very important to have effective data race detection tools.

المنارة للاستشارات
www.manaraa.com

A lot of research has gone into the problem of detecting data races in parallel programs. Data race detection techniques are mainly categorized as static, dynamic and model checking. Static race detectors analyze the programs statically and report errors without actually executing the programs [4–10]. Their drawback is that they report data races on variables when in fact there are no data races; identifying real data races from the large output becomes difficult. Model checking on the other hand produces precise results but suffers from state space explosion making it impossible to use in large systems[11–16].

Dynamic data race detectors analyze the program at runtime and so the data races reported by them are real data races. Dynamic data race detectors however can reason about only a single run [17–22]. Raman et al. created a dynamic race detector for structured parallel programs that can locate races in any schedule of the program by running the program only once using limited access history[23]. The approach necessitated data race detection on every shared memory access and checking which tasks run in parallel. The approach also did not provide any functionality to manipulate the scheduler at runtime making it unsound for programs with mutual exclusion. When accesses to shared variables are protected using mutual exclusion, different program outcomes are observed. It is necessary to analyze all possible program behaviors to ensure data race freedom.

This paper introduces an improved technique for data race detection that combines dynamic race detection for structured parallel languages with model checking to overcome the limitations of both of them. This technique makes use of computation graphs to represent the happens-before relation of the events of the program in the form of a directed acyclic graph [24]. The nodes represent the various tasks that are spawned during the program execution and store the references to shared heap locations that have been accessed by those tasks. To detect data races, the task nodes that can execute in parallel are identified in the graph and the memory locations stored in these nodes are compared to detect conflicts. For building such computation graphs, the runtime should have the ability to call-back when threads are forked or joined, and to record memory accesses on heap locations that may be shared.

المنارة للاستشارات
www.manaraa.com

The model checking part of the solution comes into play for programs with critical sections. In programs with critical sections, different computation graph structures can arise based on the order of execution of the critical sections. The technique presented here creates all such computation graphs using a scheduler that checks for critical sections and builds schedules to consider all possible computation graph structures [25]. Hence, this method is sound for all programs with a given input. Since this technique uses scheduling only on critical sections as opposed to JPF which schedules on every shared access, the state space of this technique is way smaller compared to JPF.

This paper presents an implementation of this data race detection technique for the Java implementation of the Habanero programming model. The implementation uses JPF's virtual machine for the runtime support and it uses a specialized runtime for the Habanero language that is targeted specifically for verification[14, 25]. The performance is compared with two other model checking approaches implemented by JPF: Precise Race Detector and Permission regions [11], [25]. The results show a significant reduction in the state space and time needed for verification.

Thesis Statement: A computation graph is a suitable common representation of the execution of any task parallel program. The computation graph is sufficient to determine all relevant schedules over tasks that need to be explored to enumerate all the possible behaviors of the program. Such an exhaustive enumeration is enough for verifying deterministic behavior in task parallel programs.

Main Contributions:

1. A data race detection algorithm using computation graphs that runs in $\mathcal{O}(N^2)$ time where N is number of nodes in the graph.
2. Semantics for task parallel programs that include steps for creating computation graphs.
3. Dynamic improvement to the data race detection algorithm for structured parallel programs.
4. A scheduling algorithm to create all computation graphs for programs containing mutual exclusion.
5. An implementation of the data race detection algorithm for Habanero Java.

المنارة للاستشارات
www.manaraa.com

6. An empirical study over a set of benchmarks comparing performance of the data race detection algorithm to JPF.

المنارة للاستشارات
www.manaraa.com

Chapter 2

Data Race Detection

A computation graph for a task parallel program is a directed acyclic graph that represents the execution of the program. The graph consists of nodes that denote the various parallel operations. The nodes also store references to the memory locations accessed by the tasks.

Definition 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.*

$$\omega : (N \mapsto 2^V)$$

where V is the set of the unique identifiers for the shared locations.

Figure 2.1 shows a sample computation graph. In this graph, nodes n_0, n_0', n_0'', r_1, and r_1' belong to task t_0. Task t_0 spawns two tasks t_1 and t_2. Node n_1 belongs to task t_1 and node n_2 belongs to task t_2. Node r_1 and r_1' are join nodes for tasks t_1 and t_2.

www.manaraa.com

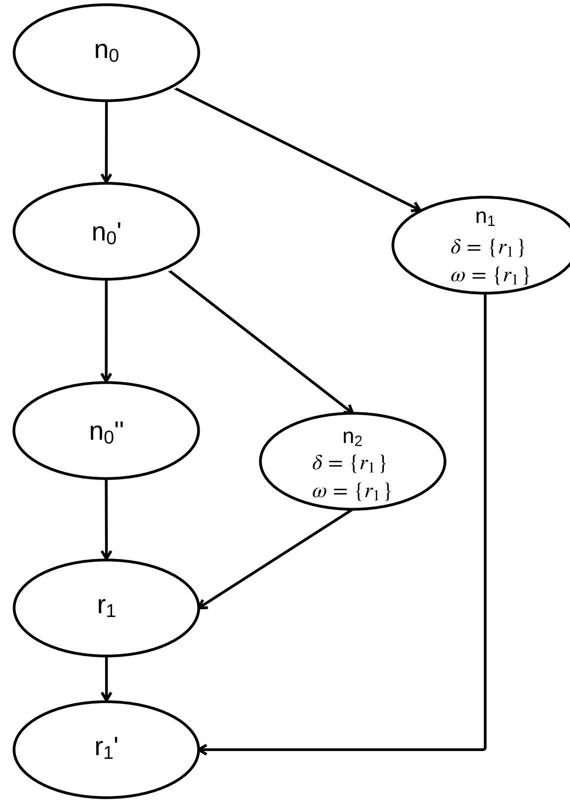


Figure 2.1: Computation Graph Example.

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** $(n \not\prec n') \wedge (n' \not\prec n)$ **then**
8: **bool** $\texttt{rw} = (\delta(n) \cap \omega(n') \neq \emptyset)$
9: **bool** $\texttt{wr} = (\omega(n) \cap \delta(n') \neq \emptyset)$
10: **bool** $\texttt{ww} = (\omega(n) \cap \omega(n') \neq \emptyset)$
11: **if** $(\texttt{rw} \vee \texttt{wr} \vee \texttt{ww})$ **then**
12: **Report Data Race and Exit**
13: **end if**
14: **end if**
15: **end for**
16: **end for**
17: **end function**

www.manaraa.com

Computation graphs can be used to detect data races in parallel programs. Every node in the computation graph represents a block of sequential operations. A computation graph is a partially ordered set of nodes that gives the relationship of the tasks in the program. The transitive closure of the graph gives the reachability of 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. The operations that may execute in parallel are unordered: $n_1 \not\prec n_2$ and $n_2 \not\prec n_1$, i. e. n_1 does not happen before n_2 and n_2 does not happen before n_1. Once these unordered nodes are identified, the memory accessed by the operations performed in these nodes is checked to detect data races.

Algorithm 1 gives the pseudo-code of the algorithm to detect data races in computation graphs. It takes the computation graph as input and reports a data race if an access violation is observed in the graph. The algorithm works as follows. The nodes in the computation graph are added to a topologically sorted set on Line 1. The ith node in the order 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 unordered. If the nodes are unordered, then the sets of memory locations accessed by each node are checked for conflict on Line 11. If any of the sets shares an element, then any one of those elements is a location where a data race occurs in the program. A data race is reported by the algorithm on Line 12. If the intersecting sets are empty, then the algorithm proceeds to check the next node until either a data race is reported or all the nodes have been verified.

Consider again the example in Figure 2.1 with the topological ordering: n_0, n'_0, n''_0, n_1, r_1, n_2, and r'_1. Node n_0 happens before all other nodes so it cannot data race with anything. The next node in the topological ordering is n'_0. It is not ordered relative to n_1 so n'_0 and n_1 are parallel. No race is reported and the analysis proceeds because there are no conflicting accesses made by these nodes. All the nodes are checked one by one in a similar way. The nodes n_1 and n_2 are unordered since there is no path from n_1 to n_2 and both are writing to variable r_1. Therefore, `ww` is set to true and a race is reported for these two nodes on r_1.

المنارة للاستشارات
www.manaraa.com

The algorithm runs in quadratic time for the number of nodes in the computation graph. The topological ordering of nodes can be done in $\mathcal{O}(\mathrm{N}^2)$. When nodes are topologically ordered, reachability of nodes can be checked in $\mathcal{O}(\mathrm{N})$ time. Therefore, the time required to check if two nodes are executing in parallel is $\mathcal{O}(\mathrm{N}^2)$. The time required to check the intersection of read or write sets of shared locations is $\mathcal{O}(m_1 + m_2)$ where m_1 and m_2 are the sizes of the two sets. $(m_1 + m_2)$ is much smaller than N. Therefore, the time complexity of Algorithm 1 is $\mathcal{O}(\mathrm{N}^2)$.

Definition 2. ***Sound:*** *A data race detection algorithm is sound if it does not miss any data race in a program for a given input.*

If the sound algorithm declares a program to be data race free, no race can exist in execution of the program for the given input on any schedule; although, it may reject programs as having data races when in fact they do not. It may under-approximate the set of data race free programs.

Definition 3. ***Complete****: A data race detection algorithm is complete if it does not report data races in programs that are data race free.*

A complete algorithm may accept programs as data race free when in fact they have data races. It may over-approximate the set of data race free programs.

Theorem 1. *Algorithm 1 is sound and complete for a given computation graph G.*

Proof. The computation graph is a directed acyclic graph. The transitive closure of the graph gives the reachibility relationship of the tasks. The transitive closure is a strict partial order over the nodes of the graph. The data race detection algorithm checks if nodes n and n' in the graph are unordered on Line 7. The statements may be executed in parallel by these nodes. The memory accessed by these tasks is compared and a race is reported if a conflict is detected on Line 12. Therefore, when algorithm 1 declares a computation graph to be data race free, no race can exist in that graph and when a race is reported by the algorithm, there definitely exists two tasks that execute in parallel and have conflicting accesses to a shared variable. Hence, Algorithm 1 is sound and complete for a given computation graph. □

المنارة للاستشارات
www.manaraa.com

Chapter 3

Computation Graphs

Bouajjani and Emmi created a formal model of isolated hierarchical parallel computation that covers many existing task parallel languages (e.g., Cilk, X10, Chapel, Habanero, etc.) [26]. Real world task parallel models are not isolated so tasks may share memory (intentionally or otherwise). This paper uses the formalism of Bouajjani and Emmi to define the construction of the computation graph from program execution but adds global variables. As before, a *region* groups tasks by storing task handles, but now each region also holds a variable that can be shared. Tasks are expanded to include access lists to denote region variables available for reading or writing.

3.1 Surface Syntax

The surface syntax for the language is given in Figure 3.1. A program **P** is a sequence of procedures. The procedure name p is taken from a finite set of names $\texttt{Proc}$. Each procedure has a single L-type parameter $\texttt{l}$ taken from a finite set of parameter names $\texttt{Vars}$. The body of the procedure is inductively defined by s. The semantics is abstracted over concrete values and operations, so

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

Figure 3.1: The surface syntax for task parallel programs.

المنارة للاستشارات
www.manaraa.com

the possible types of $\texttt{l}$ 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 $\texttt{Exprs}$. Values are given by the finite set $\texttt{Vals}$ and include at least Boolean values. $\texttt{Exprs}$ contain $\texttt{Vals}$ 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 $\texttt{Regs}$ be a finite set of region identifiers. Associated with each region r is a single variable referenced in the surface syntax by $\texttt{l}(r)$. 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 $\texttt{Regs}^*$ (i.e., the Kleene closure on $\texttt{Regs}$), 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), $\vec{r_\delta}$ 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 $\texttt{Stmts}$ be the set of all statements and let $\texttt{Rets} \subseteq (\texttt{Vals} \rightarrow \texttt{Stmts})$ 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 knows about in r finish execution. Similarly, when a task issues an **ewait** with region r, it is blocked until one task it knows about in r completes. 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*.

المنارة للاستشارات
www.manaraa.com

```
proc main (var n : int)
  n := 1;
  post r₁ ← p₁ n ε {r₁} {r₁} λv.n := n + v;
  post r₁ ← p₂ n ε {r₁} {r₁} λv.n := n + v;
  await r₁
proc p₁ (var n : int)
  l(r₁) := l{r₁} + n;
  return (n + 1)
proc p₂ (var n : int)
  l(r₁) := l{r₁} + n;
  return (n + 2)
```

Figure 3.2: A simple example of a task parallel program.

Figure 3.2 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 Figure 2.1.

3.2 Tree-based Semantics

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 valuation of the procedure local variable $\mathtt{l}$, 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.

المنارة للاستشارات
www.manaraa.com

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, $m : \texttt{Regs} \rightarrow \mathbb{M}[\texttt{Configs}]$, 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:

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

If $e[\ell/\texttt{l}, \ell(r_0)/\texttt{l}(r_0), \ell(r_1)/\texttt{l}(r_1), \ldots]$ has any free variables, then by definition, $[\![e[\ell/\texttt{l}, \ell(r_0)/\texttt{l}(r_0), \ell(r_1)/\texttt{l}(r_1), \ldots]]\!]_e$ has no meaning and is undefined (i.e., $e(t) = \emptyset$). 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 a single $\diamond$-labeled leaf, task-labeled vertexes, and region-

المنارة للاستشارات
www.manaraa.com

labeled edges. 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., $C[\langle t, m\rangle] \rightarrow C[\langle t', m\rangle]$ 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., $C[\langle T[s_1], m\rangle] \rightarrow C[\langle T[s_2], m\rangle]$ 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.

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 $\omega(n) = \emptyset$).

Figure 3.3 lists the intra-procedural transition rules. The rules omit the configuration context since intra-procedural statements do not need the region valuation from the context. The rules define the intra-procedural statements in the usual way. Of note is the update of the computation graph to record any read region variables from expressions or any write region variables from an assignment. The notation, $\delta = \delta \cup (n \mapsto \eta(e))$, is understood to update δ such that n additionally maps to $\eta(e)$.

المنارة للاستشارات
www.manaraa.com

ASSIGN LOCAL

$$\frac{\ell' \in e(\ell, \vec{r_\delta}) \qquad \delta = \delta \cup (n \mapsto \eta(e))}{\langle \ell, S[\texttt{l} := e], d, \vec{r_\delta}, \vec{r_\omega}, n\rangle \to \langle \ell', S[\textbf{skip}], d, \vec{r_\delta}, \vec{r_\omega}, n\rangle}$$

ASSIGN REGION

$$\frac{\begin{array}{l}\ell \in e(T) \qquad r \text{ is found in } \vec{r_\omega}(T) \qquad \ell(r) = \ell \\ \delta = \delta \cup (n \mapsto \eta(e)) \qquad \omega = \omega \cup (n \mapsto \{r\})\end{array}}{T[\texttt{l}(r) \ := \ e] \to T[\textbf{skip}]}$$

SKIP

$$\frac{}{T[\textbf{skip};\ s] \to T[s]}$$

ASSUME

$$\frac{\textbf{true} \in e(T) \qquad \delta = \delta \cup (n \mapsto \eta(e))}{T[\textbf{assume}\ e] \to T[\textbf{skip}]}$$

IF-THEN

$$\frac{\textbf{true} \in e(T) \qquad \delta = \delta \cup (n \mapsto \eta(e))}{T[\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2] \to T[s_1]}$$

IF-ELSE

$$\frac{\textbf{false} \in e(T) \qquad \delta = \delta \cup (n \mapsto \eta(e))}{T[\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2] \to T[s_2]}$$

DO-LOOP

$$\frac{\textbf{true} \in e(T) \qquad \delta = \delta \cup (n \mapsto \eta(e))}{T[\textbf{while}\ e\ \textbf{do}\ s] \to T[s;\ \textbf{while}\ e\ \textbf{do}\ s]}$$

DO-BREAK

$$\frac{\textbf{false} \in e(T) \qquad \delta = \delta \cup (n \mapsto \eta(e))}{T[\textbf{while}\ e\ \textbf{do}\ s] \to T[\textbf{skip}]}$$

Figure 3.3: The transition rules for the intra-procedural statements.

www.manaraa.com

CALL

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

POST

$$\frac{\begin{array}{l}n'_0 = \text{fresh}() \qquad n_1 = \text{fresh}() \\ N = N \cup \{n'_0, n_1\} \qquad E = E \cup \{\langle n_0, n'_0\rangle, \langle n_0, n_1\rangle\} \\ \ell \in e(\ell', \vec{r_\delta}') \qquad \delta = \delta \cup (n_0 \mapsto \eta(e)) \\ m' = (m \setminus m|_{\vec{r}}) \cup (r \mapsto \langle\langle \ell, s_p, d, \vec{r_\delta}, \vec{r_\omega}, n_1\rangle, m|_{\vec{r}}\rangle)\end{array}}{\begin{array}{l}C[\langle \ell', S[\textbf{post}\ r \leftarrow p\ e\ \vec{r}\ \vec{r_\delta}\ \vec{r_\omega}\ d], \vec{r_\delta}', \vec{r_\omega}', d', n_0\rangle, m] \rightarrow \\ C[\langle \ell', S[\textbf{skip}], \vec{r_\delta}', \vec{r_\omega}', d', n'_0\rangle, m']\end{array}}$$

EWAIT

$$\frac{\begin{array}{l}n' = \text{fresh}() \\ N = N \cup \{n'\} \qquad E = E \cup \{\langle n, n'\rangle, \langle n(t_2), n'\rangle\} \\ m_1 = (r \mapsto \langle t_2, m_2\rangle) \cup m'_1 \qquad s \in \text{rvh}(t_2)\end{array}}{\begin{array}{l}C[\langle \ell, S[\textbf{ewait}\ r], \vec{r_\delta}, \vec{r_\omega}, d, n\rangle, m_1] \rightarrow \\ C[\langle \ell, S[s], \vec{r_\delta}, \vec{r_\omega}, d, n'\rangle, m'_1 \cup m_2]\end{array}}$$

AWAIT-NEXT

$$\frac{\begin{array}{l}n' = \text{fresh}() \\ N = N \cup \{n'\} \qquad E = E \cup \{\langle n, n'\rangle, \langle n(t_2), n'\rangle\} \\ m_1 = (r \mapsto \langle t_2, m_2\rangle) \cup m'_1 \qquad s \in \text{rvh}(t_2)\end{array}}{\begin{array}{l}C[\langle \ell, S[\textbf{await}\ r], \vec{r_\delta}, \vec{r_\omega}, d, n\rangle, m_1] \rightarrow \\ C[\langle \ell, S[s;\ \textbf{await}\ r], \vec{r_\delta}, \vec{r_\omega}, d, n'\rangle, m'_1 \cup m_2]\end{array}}$$

AWAIT-DONE

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

Figure 3.4: The transition rules for the inter-procedural statements.

The notation $\vec{r_\omega}(T)$ in the assign-region rule is used to indicate the read-region vector in the task or task context, $T = \langle \ell, S, d, \vec{r_\delta}, \vec{r_\omega}, n\rangle$. Similar notation is used in other rules to access the tuple.

Figure 3.4 shows semantics for the inter-procedural statements. 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 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

المنارة للاستشارات
www.manaraa.com

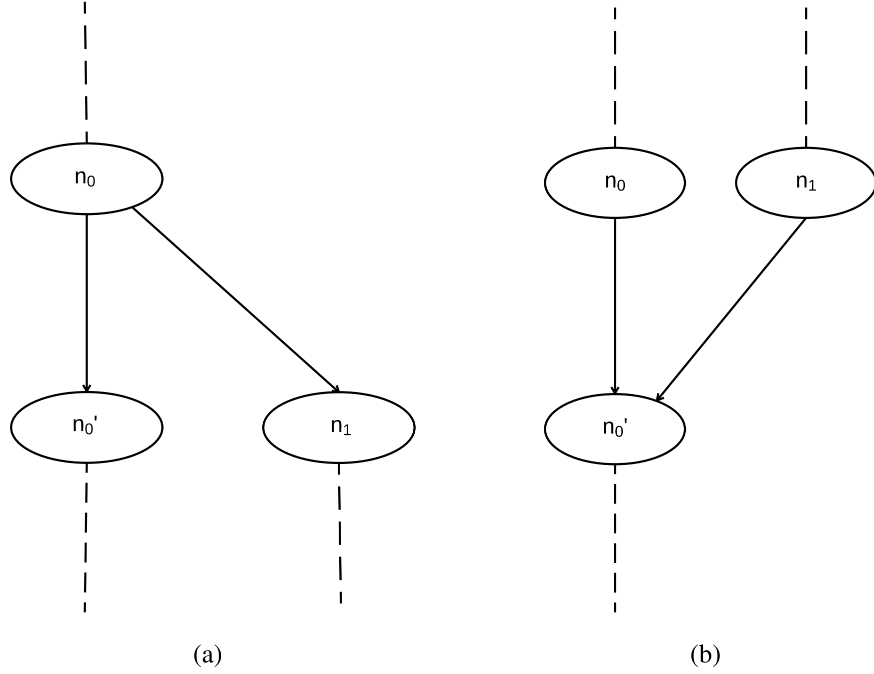


Figure 3.5: Steps involved in computation graph creation.

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 Figure 3.5(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 executed by task 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 Figure 3.5(b). The configuration $(r \mapsto \langle t_2, m_2 \rangle)$ is removed from the region valuation m of task t_1. After the transition, the current node of task t_1 is

المنارة للاستشارات
www.manaraa.com

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 whose handles are stored in region r that the task t_1 owns are executed to completion. 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. Similar to EWAIT, 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 Figure 3.5(b) and the current node of task 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 Figure 3.2 is presented in Figure 2.1. When the program starts executing, a node n_0 is added to the graph to represent the procedure $main$. When the task t_1 is posted by the procedure $main$ to execute procedure p_1, two new nodes n'_0 and n_1 are added to the graph to represent the statements executed by the procedure $main$ and procedure p_1 respectively. Similarly, when task t_2 is posted by the procedure main to execute procedure p_2, two new nodes n''_0 and n_2 are added to the graph. When the main task calls **await** on r_1, its execution is suspended until t_1 and t_2 finish execution. When the **await** is executed, node r_1 and r'_1 are added to the graph. The read and write to region variable r_1 by the tasks t_1 and t_2 is updated in nodes n_1 and n_2 using the functions δ and ω respectively.

The order of synchronization of tasks t_1 and t_2 affects the value of the variable n in the $main$ task. The return value handlers of the tasks get executed in different orders under different 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 n is 2.

Theorem 2. *The computation graph represents the correct ordering of events in a program and stores the accesses to shared variables in the program. The sequential events are ordered while the concurrent events are unordered.*

www.manaraa.com

Proof. Proof by definition: There are two types of operations performed in a task parallel program: inter-procedural and intra-procedural. The inter-procedural statements create different nodes in the graph and are responsible for maintaining the correct ordering of events in the program. The nodes in the computation graph contain read/write sets to store the accesses to shared variables by the tasks. The intra-procedural operations do not affect the structure of the computation graph; however, they update the read/write sets of the nodes in the computation graph when the tasks access shared variables. These operations are discussed separately below.

The semantics for inter-procedural statements are given in Figure 3.4. The inter-procedural statements are **post**, **ewait**, **await** and **call**. When a **post** statement is executed, the POST rule is fired. It creates two new nodes in the computation graph. One node represents the statements executed by the newly posted task and the other node represents the statements executed by the calling task immediately following the **post** statement. These nodes are set as the active nodes for these tasks. Any access to the shared memory is stored in the read/write sets for the active node. These two nodes are unordered since the statements are executed concurrently by these tasks.

The **ewait** is used to synchronize a child task with its parent task. The EWAIT rule creates a join node in the computation graph. Both the child and the parent task's active nodes are connected to the join node. The added edges order this node after the active nodes in the child task and the parent task. The **await** statement joins all the children tasks posted in a region to the parent task. The **await** statement fires the AWAIT-NEXT rule that joins one child task at a time to the parent task. Similar to the EWAIT rule, AWAIT-NEXT also creates a join node for every child task. The join node is set as the active node for the calling task. Any shared memory accesses by the calling task are registered in the read/write sets of the active node.

Finally, the **call** is semantic sugar for a **post** followed by an **ewait**. As such, even though the calling task gets a new active node to reflect its concurrent relationship to the newly created task, the read/write sets in that node are never updated since the calling task executes **ewait** immediately after the **post**, which does not read/write any region variables, and once the **ewait** completes, the task gets a new active node ordered after the join from the task created by the call.

المنارة للاستشارات
www.manaraa.com

The intra-procedural statements are **assign**, **skip**, **assume**, **if-then-else** and **do-while**. The semantics for intra-procedural statements is given in Figure 3.3. The semantic rules for intra-procedural events show that they do not change the structure of the computation graph since none of the rules create any new nodes or edges in the computation graph.

The SKIP rules does not interact with any shared variables in the program. The **if-then-else** and **do-while** statements fire IF-THEN, IF-ELSE, DO-LOOP and DO-BREAK rules. These rules only read shared variables. Therefore, only the read sets for the active nodes set by the inter-procedural statements are updated by the statements. The ASSIGN LOCAL rule only updates the read set of the active node since this rule does not update any shared variables. Whereas the ASSIGN REGION rule updates both read/write sets of the active node, since shared program variables are updated by this rule. As such, by definition, the computation graph exactly reflects the orders defined by the semantics and only updates read/write sets that are defined by the semantics. □

Corollary 1. *Applying Algorithm 1 to computation graphs created using the semantics of task parallel programs is complete for data race detection in the given program input – data race free programs will never be rejected; but, programs with data race may be accepted because the data race did not manifest in the computation graph from the executed schedule.*

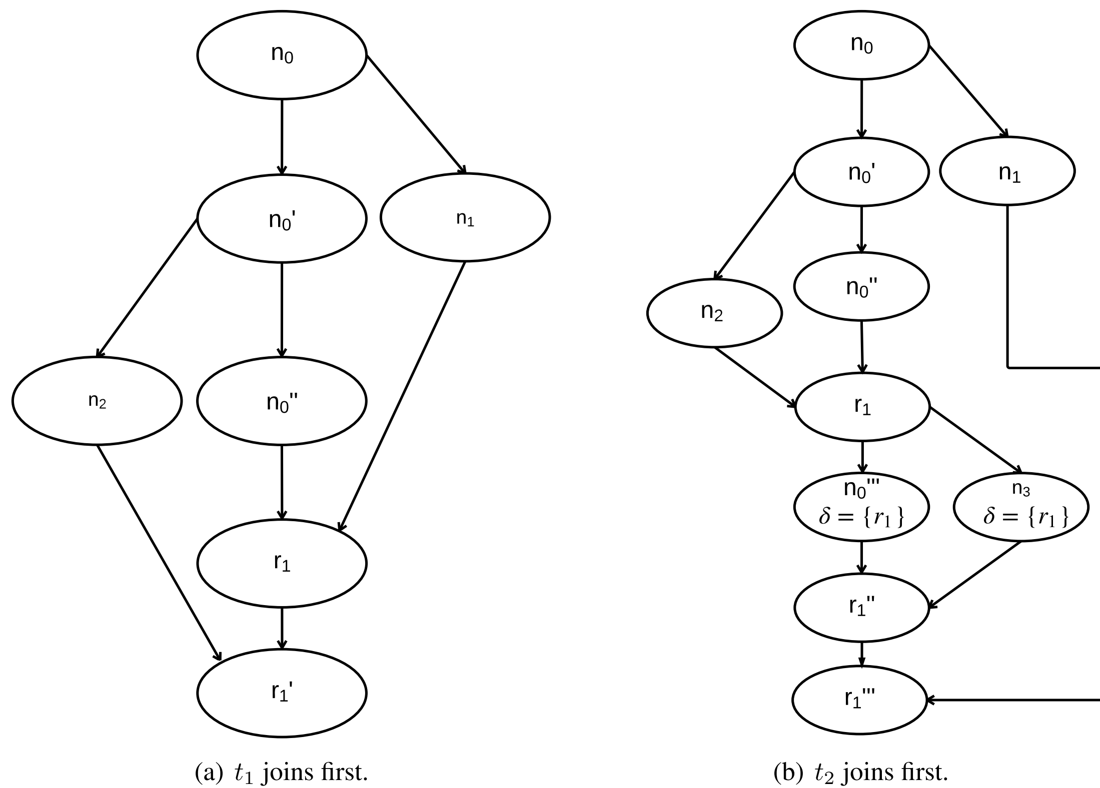
Proof. Proof by example: A task parallel program can have different computation graphs based on the schedule followed by the tasks during the program execution. If Algorithm 1 does not report a race for a computation graph obtained from some execution of the program, data races may still be present under some other program schedule.

Consider the example in Figure 3.6. The task parallel program in the example has a data race under one program schedule and it is data race free under a different schedule. The computation graphs for the different schedules are shown in Figure 3.7. If the program follows the first schedule, (i.e., task t_1 joins before t_2) task t_3 is not spawned and there is no data race in the program. If the program, however, follows the second schedule(i.e., task t_2 joins first), then a new task t_3 is created by task t_0 and there is a data race on region variable r_1.

المنارة للاستشارات
www.manaraa.com

```
proc main(var n : int)
  n := 1;
  post r1 ← p1 n ε {r1} {r1} λv.n := v;
  post r1 ← p2 n ε {r1} {r1} λv.n := v;
  ewait r1
  if (n == 1) then
     post r1 ← p3 n ε {r1} {r1} λv.n;
     r1 := 1
  await r1
proc p1(var n : int)
  return 0
proc p2(var n : int)
  return 1
proc p3(var n : int)
  r1 := 2
```

Figure 3.6: Parallel program with different computation graphs under different schedules.



(a) t_1 joins first.

(b) t_2 joins first.

Figure 3.7: Computation graphs of example in Figure 3.6.

المنارة للاستشارات
www.manaraa.com

Theorem 1 shows that Algorithm 1 is sound and complete for a given computation graph. When Algorithm 1 is applied to task parallel programs with a given input, it may accept programs as data race free that in reality contain data races. This is evident from example in Figure 3.6. Therefore, determining data race freedom from a single schedule of a task parallel program using Algorithm 1 is complete for the input program because different schedules create different computation graphs.

□

المنارة للاستشارات
www.manaraa.com

Chapter 4

Structured Parallel Languages

Non-determinism arises in task parallel programs primarily due to two reasons: data races and the order in which return value handlers are executed. Return value handlers act on local variables whereas data races occur in shared variables. Non-deterministic programs create different computation graphs under different program schedules. When the behavior of the program is non-deterministic, the result of the data race detection algorithm for a computation graph can be applied only for that particular program run. The non-determinism in program behavior due to different order of execution of return value handlers is countered by structured parallel languages such as Habanero Java and X10 by imposing an order on the task synchronization when the return value handlers do not commute.

These languages impose the following restrictions to ensure determinism in program behavior in the absence of data races:

- 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 $\langle$T[**post** $r_0 \leftarrow p_0\ e\ \varepsilon\ \vec{r}\ \vec{r}$ λv.v; **await** r_0; **await** $r_1; \ldots], m_0\rangle$ on some procedure p_0, $\vec{r}$ is the region sequence containing all regions and $\forall r \in \texttt{Regs}, m_0(r) = \emptyset$

المنارة للاستشارات
www.manaraa.com

```
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 4.1: Example of an HJ Program.

4.1 Habanero Java

Habanero Java is a structured parallel programming language that gives importance to the usability and safety of parallel constructs. It guarantees properties such as determinism and serialization for subsets of parallel constructs. However, these guarantees hold only in the absence of data races. It provides various parallel constructs to create structured parallel programs.

Figure 4.1 shows an example of an HJ program. The main task has two nested finish-blocks with tasks being posted to both these blocks and a future task.

The **async** construct creates a new asynchronous task that runs in parallel with the parent task. Task passing is not allowed in HJ, so the sequence of regions whose handles are passed to the child task is empty (ε). The newly created task has read and write access to all the region variables in the program.

المنارة للاستشارات
www.manaraa.com

```
proc main (var n : int)
  l(r_1) := 0;
  post r_1 ← p_1 0 ε r⃗ r⃗ λn.n;
  post r_2 ← p_2 0 ε r⃗ r⃗ λn.n;
  await r_2;
  await r_1;
  post r_3 ← p_2 0 ε r⃗ r⃗ λn.r_1 := n;
  ewait r_3;
proc p_1 (var n : int)
  l(r_1) := l(r_1) + 1
proc p_2 (var n : int)
  l(r_1) := l(r_1) + 2
proc p_3 (var n : int)
  return 5
```

Figure 4.2: HJ program converted to task parallel language.

A **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. Each **finish** construct creates a new region to post tasks. Every task has a unique immediately-enclosing-finish (IEF) during program execution. That IEF is the innermost finish construct containing the task. The runtime holds stacks of finish-blocks. Every stack is associated to a task to track the nesting of finish-blocks in this task. When a task is created, it is added to the parent task's active finish-block. In this way, when a parent reaches the end of a **finish** construct, it calls **await** on the region belonging to this finish-block to join on all tasks in the current finish-block. After joining, the finish-block is popped off the stack.

The **future** construct lets tasks return values to other tasks: **future** f = **async** s creates a new child task to execute s. The local variable f contains a handle to the newly created task that can be used to obtain the value returned by s. The blocking operation $f.get()$ retrieves this value when the child task completes execution.

Figure 4.2 shows the conversion of program from example Figure 4.1 to the generic task parallel language in this paper. The procedure $main$ posts task from the outer finish block to region r_1 and task from the inner finish block to region r_2. Since, the inner finish block completes execution

www.manaraa.com

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.

Habanero also includes loop parallelism constructs such as *forasync* and *forall* which are syntactic sugar for the presented constructs. An implicit finish is included at the end of forall iterations whereas forasync iterations do not have an implicit finish.

4.2 Properties of structured parallel programs

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

Lemma 1. *For a graph* $G \in \mathcal{G}(P), \mathrm{DRF}(G) \rightarrow \{\forall G' \in \mathcal{G}(P), \mathrm{DRF}(G')\}$.

Proof. Suppose there exists $\{G, G'\} \subseteq \mathcal{G}$ such that $\mathrm{DRF}(G)$ is true but $\mathrm{DRF}(G')$ is false. As such, either G and G' have the same structure and differ in the region variables accessed, or they have different structures all together. To accomplish either situation, there must be a source of non-determinism either in the program P itself or as a result of the semantic definition for task parallel programs and how computation graphs are derived from executions. Since the input to the program is fixed and expression evaluation is deterministic by definition (e.g., the choice operator is not allowed), the non-determinism needed to create G and G' must arise through task interaction.

Tasks interact at creation, completion, and through shared region variables. The interaction needs to be such that it causes a task in P to follow a different control flow path to access different region variables or to post and synchronize tasks differently in order to create G and G' so that one has no data race while the other one does. At task creation, the POST rule in Figure 3.4 indicates that the parent task passes to the child task the value of the child's local variable, other tasks from the parent, the read and write region variables, and the return value handler. Each is discussed separately.

Structured parallel languages do not allow task passing by definition. The definition also mandates all regions for reading and writing in each task. As such, no different information is

المنارة للاستشارات
www.manaraa.com

exchanged that can lead to G and G' by task passing or access lists—synchronization between tasks (e.g., **await** and **ewait**) and available regions to access are identical. That leaves the child's local variable and the return value handler to discuss.

Structured parallel languages by definition restrict side-effecting return value handlers (i.e., handlers that alter the local variable in the parent task) to appear in the **ewait** statement only, and it further restricts that the statement indicate the task for which it is to wait. This restriction effectively serializes the computation in the return value handlers to always be deterministic (i.e., it follows the same order to yield the same computation, in the absence of data race, since expressions are deterministic). Further, since the definition restricts return values handlers for the **await** statement to not side-effect, it is not possible to create G and G' with return value handlers in the absence of data race—task completion is ordered by **ewait** and it does not matter for **await**.

Turning to the child's local variable, to create G and G', some task in the program P must see a different value for that local variable which is then used in an expression such that the same task takes one control path in G and a different control path in G'. The only way to alter the value of a local variable is through a conflicting access on some region variable shared between two tasks (e.g., a data race), but since that does not exist in G, $\mathrm{DRF}(G)$ is true, then it cannot exist in G' either because the program P is deterministic by virtue of G being data race free—a contradiction. □

Lemma 1 proves the claimed property that structured parallel programs in the absence of data race are deterministic [3]. And is the first formalization of that property. The other claimed properties can be derived from Lemma 1 but are not part of this paper.

Corollary 2. *For a graph* $G \in \mathcal{G}(P), \neg\mathrm{DRF}(G) \rightarrow \{\forall G' \in \mathcal{G}(P), \neg\mathrm{DRF}(G')\}$.

Proof. Trivial from Lemma 1. □

Theorem 3. *Algorithm 1 is sound and complete for structured parallel programs with fixed input.*

Proof. From Lemma 1 and Corollary 2, it can be seen that a single computation graph is enough to verify a structured parallel program under any schedule. Theorem 1 states that Algorithm 1 is sound

المنارة للاستشارات
www.manaraa.com

and complete for a computation graph. Therefore, Algorithm 1 is sound and complete for structured parallel programs with fixed input. □

المنارة للاستشارات
www.manaraa.com

Chapter 5

On-the-fly data race detection

The data race detection technique presented in this work performs the analysis after the program has finished execution (post-mortem analysis). To improve the efficiency of analysis at run time, this paper presents a dynamic improvement for structured parallel programs. This technique is called on-the-fly data race detection.

The data race detection is run on a region as soon as await finishes execution on that region (i.e., AWAIT-DONE fires). If no race is reported, all the nodes belonging to that region are merged into an equivalent master node that represents the region. The transformation preserves the partial order relative to other tasks. The variables accessed by the tasks in the region are added to the master node.

To implement this technique, the AWAIT-DONE rule has been modified as follows. The join and meet nodes for set of nodes connected to the current node of task executing AWAIT-DONE are identified.

Definition 4. Join : *In a partially ordered set* $(N, \preceq)$*, an element* n_0 *is a* join *of two unordered elements* n_1 *and* n_2 *if the following conditions hold:*

- $n_0 \prec n_1$ *and* $n_0 \prec n_2$ *(i.e.,* n_0 *happens before* n_1 *and* n_2*).*
- *For any element* n *in* N*, such that* $n \prec n_1$ *and* $n \prec n_2$*, we have* $n \prec n_0$*.*

A join is the least upper bound for a subset of elements in the partially ordered set. Conversely, a meet is greatest lower bound for the elements in the partially ordered set. Join and meet are symmetric duals with respect to order inversion.

المنارة للاستشارات
www.manaraa.com

Definition 5. Meet : *In a partially ordered set* $(N, \preceq)$*, an element* n_0 *is a* meet *of two unordered elements* n_1 *and* n_2 *if the following conditions hold:*

- $n_1 \prec n_0$ *and* $n_2 \prec n_0$ *(i.e.,* n_0 *happens after* n_1 *and* n_2*).*
- *For any element* n *in* N*, such that* $n_1 \prec n$ *and* $n_2 \prec n$*, we have* $n_0 \prec n$*.*

Definition 6. *A* lattice *is defined as a partially ordered set such that every pair of elements has a unique* join *and a unique* meet.

Lemma 2. *A computation graph of a structured parallel program is a* lattice.

Proof. Structured parallel programs have a restriction of joining all unsynchronized tasks to the main task at the end of program execution. The node that represents the main task when the program starts execution acts as join node for all the nodes in the graph and the last await node acts as meet node. For any pair of nodes n_1 and n_2, either the nodes are ordered or unordered. If the nodes are ordered, $n_1 \prec n_2$ or vice versa, then there is a unique meet or join depending on the ordering. For example, if $n_1 \prec n_2$, then n_1 is the join and n_2 is the meet.

If, however, the nodes are unordered, then it needs to be proven that the meet and join are unique based on the structure of the graph. Consider the rules that add nodes to the graph. Nodes are added when POST, AWAIT-NEXT or EWAIT rule fires. A POST rule is fired when a new task is created. The parent task creates branching in the graph by adding nodes to represent the parent and child task that executes in parallel. Since every task can only have a single parent, the join node for the unordered nodes is unique. Tasks synchronize only on completion when AWAIT-NEXT or EWAIT rule fires. Since structured parallel programs do not allow task passing, tasks are bound to join to their parent/ancestor task only. Therefore, they cannot be cross-edges in the computation graph. Hence, a pair of unordered nodes can have only a unique meet. Therefore, a computation graph of a structured parallel program is a lattice. □

The modified AWAIT-DONE rule for on-the-fly data race detection is presented in Figure 5.1. subGraph takes a set of nodes, finds the meet/join, and then extracts everything, inclusive, as

المنارة للاستشارات
www.manaraa.com

$$\textsc{Await-done}\quad \frac{m(r)=\emptyset \quad G'=\text{subGraph}(\forall n', \langle n', n\rangle \in E) \quad DRF(G')=true \quad n_1=\text{fresh}() \quad N = N\setminus N' \cup \{n_1\} \quad E=E\setminus E' \quad \delta(n_1)=\bigcup_{n'\in N'}\delta(n') \quad \omega(n_1)=\bigcup_{n'\in N'}\omega(n')}{C[\langle \ell, S[\textbf{await } r], \vec{r_\delta}, \vec{r_\omega}, d, n\rangle, m] \rightarrow C[\langle \ell, S[\textbf{skip}], \vec{r_\delta}, \vec{r_\omega}, d, n_1\rangle, m]}$$

Figure 5.1: AWAIT-DONE rule updated for on-the-fly data race detection.

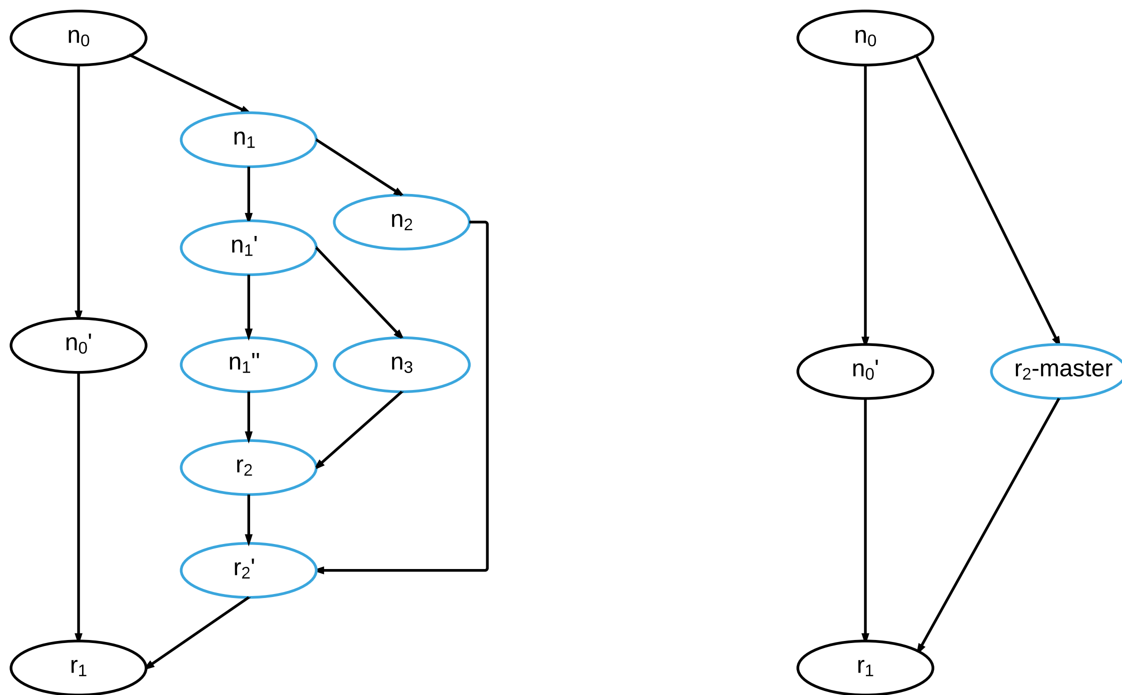
```
proc main(var n : int)
  n := 1;
  post r1 ← p1 n ε {r1} {r1} λv.n := n + v;
  await r1
proc p1(var n : int)
  post r2 ← p2 n ε {r1} {r1} λv.n := n + v;
  post r2 ← p3 n ε {r1} {r1} λv.n := n + v;
  await r2
proc p2(var n : int)
  l(r2) := n
proc p3(var n : int)
  l(r2) := n
```

Figure 5.2: A parallel program with nested regions.

a sub-graph. The sub-graph for the set of nodes connected to n is extracted using subGraph. This sub-graph is verified using Algorithm 1. If this sub-graph is data race free, the nodes in this sub-graph are deleted from the computation graph. A new master node n_1 is added to the graph in place of this sub-graph. The region variables accessed by the nodes in the sub-graph are added to n_1.

Figure 5.2 shows an example of a parallel program with nested regions. The $main$ task spawns a task t_1 in region r_1. Task t_1 spawns two new tasks t_2 and t_3 in region r_2. The await on r_2 is executed before await on r_1 making the regions nested. All the tasks have read/write access to region variable r_1. As soon as the await on region r_2 finishes execution, on-the-fly analysis is run on this region to check for data races in the nodes belonging to this region. If a race is not reported, a master node is added to the graph that represents region r_2 and the program is executed further. The

المنارة للاستشارات
www.manaraa.com



(a) Comp graph for parallel program with nested regions (b) Comp graph with master node

Figure 5.3: Computation graph of example in Figure 5.2.

المنارة للاستشارات
www.manaraa.com

computation graph for this example is shown in Figure 5.3(a). The nodes that are used to find the meet and join are r_2 and n_2. The nodes highlighted in blue denote the sub-graph that is replaced by a master node if the region is data race free. Figure 5.3(b) shows the computation graph with a master node inserted in place of region r_2.

المنارة للاستشارات
www.manaraa.com

Chapter 6

Mutual Exclusion

Data races in parallel programs lead to non-deterministic behavior of programs. Therefore, data races are termed as errors. When access to a shared variable is protected by a lock, there is a programmer intended race on the variable. The behavior of the program is non-deterministic if the programmer does not order the way in which tasks access this shared variable, but, this behavior is expected by the programmer.

Programs with data races can have different computation graph structures based on the schedule of tasks in the runtime (i.e., different update orders may result in different control flow paths). Similarly, when shared memory accesses are protected using locks, different computation graph structures can be observed based on the order in which the tasks access the protected shared variable. To ensure that a program in which tasks have mutually exclusive access to some shared variable does not have any unintended race, all the computation graph structures that can result from different schedules over the protected shared variable access have to be enumerated and analyzed.

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 how the computation graph is impacted is in Figure 6.1. 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 $\mathrm{canIsolate}(C)$ 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

المنارة للاستشارات
www.manaraa.com

ISOLATED

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

ISOLATED-NESTED

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

ISOLATED-END-NESTED

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

ISOLATED-END

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

Figure 6.1: The transition rules for isolated statements.

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.

On-the-fly data race detection is modified for programs with isolated regions. When AWAIT-DONE fires, the runtime checks if the region contains any task containing **isolated** statements. If the region does not have any isolated-nodes, data race detection is run on the region and if the region is data race free, it is replaced with an equivalent master node. If the region contains isolated-nodes,

المنارة للاستشارات
www.manaraa.com

the program execution proceeds normally (i.e., on-the-fly data race detection is not executed on the region).

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 ∈ 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
```

Algorithm 2 presents a scheduling algorithm to explore different computation graph structures in parallel programs with isolated blocks. This algorithm is adapted from the scheduling algorithm used for model checking HJ programs using permission regions [25]. The algorithm considers a simplified state of the program with `Regs` as the set of region variables that are shared among the tasks, `Tasks` is the set of tasks and t is a task. R is the set of runnable tasks.

The algorithm implements sequential semantics where only a single task is running at any time, and that task runs until it completes or isolates at which time a scheduling choice is made. Sequential semantics can be used for computation graph creation since Lemma 1 proves that the creation of computation graph is independent of the schedule that was followed to create the graph in the absence of data-race. And Corollary 2 shows that if a data-race exists, then it manifests on every schedule.

Line 2 updates the region variables and pool of tasks by running task t until it exits, 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 blocked and all the tasks that are runnable are scheduled by

المنارة للاستشارات
www.manaraa.com

```
proc main(var n : int)
  n := 1;
  post r1 ← p1 n ε {r1} {r1} λv.n := n + v;
  post r1 ← p2 n ε {r1} {r1} λv.n := n + v;
  await r1
proc p1(var n : int)
   isolated  l(r1) := n + 1
proc p2(var n : int)
   isolated  if (l(r1) = n) then
       post r1 ← p3 n ε {r1} {r1} λv.n := n + v;
   else
       l(r1) := n − 1
proc p3(var n : int)
  l(r1) := n + 2
```

Figure 6.2: Parallel program with mutual exclusion.

the runtime. When the task exits, a task is randomly selected from the set of runnable tasks and scheduled by the runtime.

For the example in Figure 6.2, two different computation graph structures can be formed based on the order of execution of isolated blocks. The computation graphs are shown in Figure 6.3. If the scheduler runs the isolated section of task t_1 first, the computation graph in Figure 6.3(a) is formed. Task t_1 changes the values of shared variable r_1 to 2. Hence, when task t_2 executes its isolated section, the if-condition fails and an additional task is not spawned by t_2. If the scheduler runs task t_2 first, the computation graph of Figure 6.3(b) is formed. In this schedule, task t_2 executes its isolated section first. Since the value of variable r_1 is 1, the if-condition is met and a new task is created by t_2.

Theorem 4. *Algorithm 2 finds all unique computation graphs for structured parallel programs with isolated sections making it sound and complete with Algorithm 1.*

Proof. Theorem 3 states that Algorithm 1 is sound and complete for structured parallel programs that do not contain isolated sections. If mutual exclusion is present, Algorithm 1 does not remain sound since different computation graph structures can be formed for such programs. The different

المنارة للاستشارات
www.manaraa.com

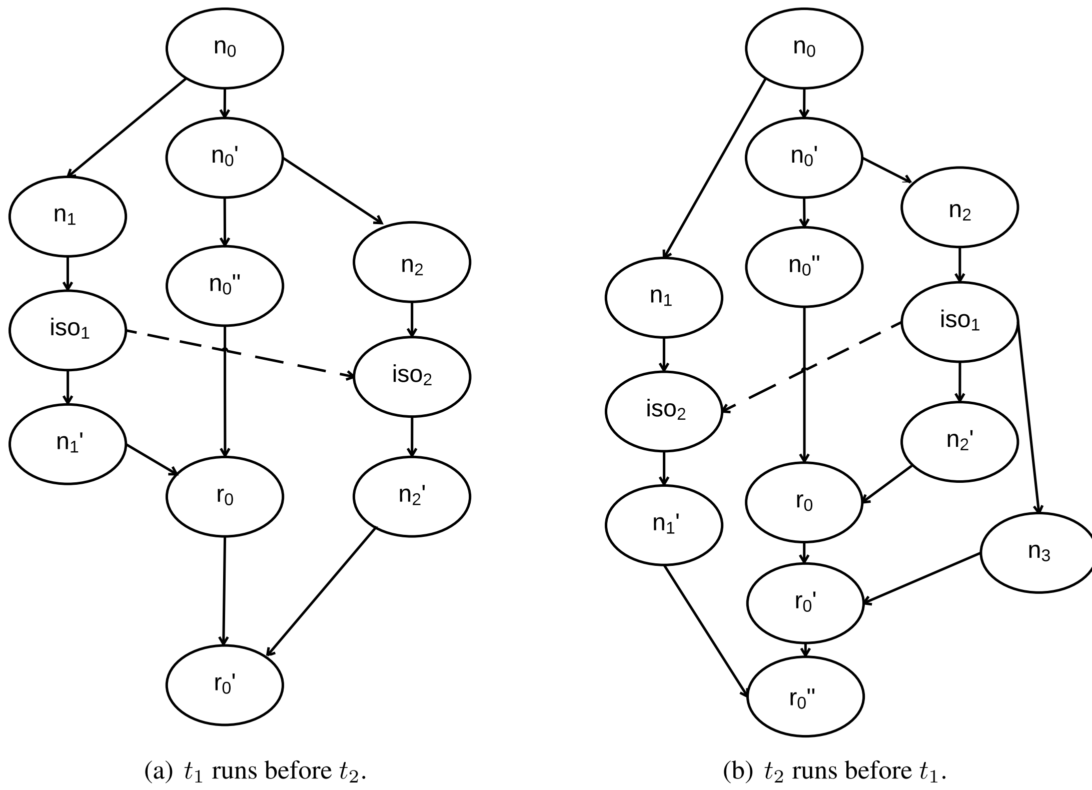


Figure 6.3: Computation graphs of example in Figure 6.2.

المنارة للاستشارات
www.manaraa.com

computation graph structures arise because the critical sections of tasks are executed in different orders under different program schedules. Algorithm 2 creates thread scheduling choices at the boundary of a critical section considering all the runnable threads that are present at the execution of a critical section. Hence, all relevant computation graphs are considered by Algorithm 1. Therefore, the data race detection using Algorithm 1 becomes sound and complete when it is used along with Algorithm 2 for structured parallel programs that have mutual exclusion. □

المنارة للاستشارات
www.manaraa.com

Chapter 7

Implementation and Results

7.1 Implementation

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 [14]. This runtime makes use of JPF to schedule and run the programs. The computation graphs are stored in a directed acyclic graph. The JGraphT library provides an implementation of directed acylic graph [27]. This library has been used to store the computation graphs. The computation graphs are exported in the dot file format for convenience and as a way to understand the structure of the program. The implementation is written in Java. It consists of 5 classes with 1600 lines of code.

JPF is modified by removing its default scheduling factory that inserts choices on all thread actions and accesses to shared variables. Instead, a new scheduling factory based on Algorithm 2 is employed for scheduling.

JPF's VM listeners have been used to keep track of various program events. The methods `objectCreated` and `objectReleased` are used to create nodes in the computation graph. The `objectCreated` method is used to track the creation of new **async** tasks. The POST rule is used to add nodes to the computation graph when the `objectCreated` method returns a task object. Similarly, the `objectReleased` method is used to track when **finish** blocks complete execution. The AWAIT-NEXT rule 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. The current node of the task calling `executeInstruction` method is updated

المنارة للاستشارات
www.manaraa.com

with the location accessed by the task during the execution of that instruction. Algorithm 1 is used to analyze the computation graphs of the program. When an array is accessed by the program, every element of the array is treated as a single variable for data race detection.

7.2 Results

The results for this technique have been compared to JPF's precise race detector and gradual permission regions based race detector on benchmarks that cover a wide range of functionality in HJ. The results show a significant improvement in the time required for verification. These two approaches were specifically chosen for comparison since the results generated by these approaches are sound for a given input just like the technique discussed in this paper.

The precise race detector explores all potential executions in a systematic way. Each execution is a sequence of transitions. Each transition takes the system from one state to another. Each transition consists of a sequence of bytecode instructions. JPF groups bytecode instructions such that an instruction that manipulates a shared variable is the first one of a transition. In every state that JPF visits, the precise race detector checks all actions that can be performed next. If this collection of actions contains at least two conflicting accesses of a shared variable, then a race on the shared variable is reported.

Gradual Permission regions use program annotations to reduce the state space of the program. Whenever a shared variable is accessed by two or more tasks in the program, the accesses have to be annotated to inform the data race detector to create different schedules for these accesses. Since the method requires manual annotation of the programs, it is prone to human errors. If the program is annotated incorrectly, the results of data race detection analysis do not have any significance.

المنارة للاستشارات
www.manaraa.com

Table 7.1: Benchmarks of HJ programs: Computation graphs vs Permission Regions vs. PreciseRaceDetector.

			Computation graphs			Permission Regions			Precise Race Detector		
Test ID	SLOC	Tasks	States	Time	Error Note	States	Time	Error Note	States	Time	Error Note
Primitive Array No Race	29	3	5	00:00	No Race	5	00:00	No Race	11,852	00:00	No Race
Primitive Array Race	39	3	5	00:00	Race	5	00:00	Race	220	00:00	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	1013102	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
Substring Search	83	59	64	00:03	Race	8	00:00	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
Reciprocal Array Sum	58	2	4	00:08	Race	32	00:06	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

المنارة للاستشارات
www.manaraa.com

Table 7.1 presents the results of verification of HJ benchmarks using computation graphs based data race detector described in this work, permission regions based extension of JPF and precise race detector extension of JPF. The number of states explored by JPF and time required for verification by each of these methods were compared. The tests were run for a maximum of an hour before they were terminated manually. For the tests that did not finish execution within the stipulated time or the ones that ran out of JVM heap memory were considered failed and marked as N/A in the table. The error note column shows the results of verification. The tests that produced erroneous results were marked with an asterisk ($*$).

The benchmarks used in this study make use of various constructs of HJ for achieving task parallelism. They spawn a wide range of tasks with smaller programs having 3-15 tasks going all the way upto 525 tasks for larger tasks. The experiments were run on a machine with an Intel Core i5 processor with 2.6GHz speed and 8GB of RAM. The number of cores used by the system is not relevant since JPF runs only a single thread at a time.

The Precise race detector inserts choices in the scheduler for all thread actions such as thread creation, synchronizations, locks etc. Therefore, it 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 is because in precise race detector, the access on an array object looks like a data race since it is not able to see the difference in the indexes.

The Gradual Permission regions based detector works pretty well compared to precise race detector. The number of states explored and time required for data race analysis by Permission regions and computation graphs is almost the same when the tasks don't access shared variables outside isolated blocks. When there are accesses to shared variables, the state space of permission regions grows very fast since the shared variable accesses have to be annotated with regions and race detector creates scheduling choices at every region boundary. Analyzing a single computation graph for a program is enough for a program without isolated blocks since by Lemma 1, if a computation graph of a program is data race free, all computation graphs from all schedules are data

المنارة للاستشارات
www.manaraa.com

Table 7.2: Comparison of results for on-the-fly and normal data race detection using computation graphs.

Num of Tasks (n)	On-the-fly		Normal	
	States	Time	States	Time
5	5	00:01	14	00:01
50	5	00:01	61	00:03
100	5	00:01	112	00:05
500	5	00:01	513	02:25
1000	5	00:01	1013	08:34

race free. Therefore, computation graphs are built using a single program schedule. This difference can be seen in examples for Add, Scalar multiply and Prime number counter benchmarks. These benchmarks use shared variables that have to be enclosed within regions which results in a large state space for permission regions and longer analysis time.

Table 7.2 presents the improvement offered by on-the-fly analysis over normal data race detection using computation graphs. The example in Figure 7.1 is used to demonstrate the difference in performance of these techniques under different program sizes.

The example in Figure 7.1 is implemented in HJ. It consists of a **finish** block with two **async** tasks that have a data race on static variable x. This **finish** block is followed by a **forall** loop which is used to control the size of the program. Note that a **forall** loop has an implicit **finish** block and each of the iterations of the loop are executed by creating **async** tasks inside the **finish** block.

From the results in table 7.2, it can be observed that the time required to analyze the program using normal data race detection using computation graphs increases as the size of the program increases. For the on-the-fly data race detection, the race detection is run on a **finish** block as soon as the **finish** block completes execution. Since a data race is present in the first **finish** block itself, the entire program is not executed. Therefore, the time required for analysis is very low.

المنارة للاستشارات
www.manaraa.com

```
public class Example{
    static int x = 0;
   public static void main(String[] args) {
      finish {
         async { //Task1
            x = x + 1;
         }
         async { //Task2
            x = x + 2;
         }
      }
      forall (1, n, (index)){
         x = x + index;
      }
   }
}
```

Figure 7.1: An HJ Program for comparing on-the-fly with normal data race detection using computation graphs.

المنارة للاستشارات
www.manaraa.com

Chapter 8

Related Work

Different types of data race detection techniques have been developed. The static race detectors analyze the source code to detect races. The dynamic race detectors use information from the actual program executions for data race detection. Another technique for data race detection is model checking. In this method, a model of the system being analyzed is created and whether this model meets the specifications is exhaustively checked.

Static data race detectors require program instrumentation by the users. They can reason about all possible program runs. The major drawback of these systems is that they produce a large number of false-positives. [4–10].

Dynamic race detectors use different techniques to detect data races at runtime. The lock-set based tools track the set of locks held by each task during execution. These sets are then used to determine conflicts over shared memory references [18, 28–30].

Dimitrov et al. developed a dynamic commutativity race detector [31]. It uses vector clocks along with a commutativity specification to generate a structural representation of parallel programs that is used to locate races. Dynamic race detectors based on hashing assert if different runs of a parallel program with the same input produce different outputs [32].

Lamport defined the happens-before relation in parallel programs [33]. The happens-before relation defines a partial order among all the operations in all the threads of a parallel program. The happen-before relation has been used in various data race detection techniques [17, 19, 20, 34–36]. This approach has also been applied to task parallel languages such as Cilk and X10 [21, 22].

المنارة للاستشارات
www.manaraa.com

Another algorithm based on the happens-before relation, discussed in the introduction, has been developed for HJ programs [23].

Model checking systematically explores the entire state space of the programs to detect concurrency issues [11–13]. The major drawback of model checking is the explosion in the state space as the program size increases. This technique has been extended to verify various task parallel languages such as HJ, X10 and Chapel[14–16]. As opposed to model checking, predictive analysis observes only a single program execution and generalizes the verification results to all possible schedules. This approach has been applied to detecting communication deadlocks in MPI programs [37].

Various methods have been developed to tackle the state explosion problem of model checking. Rely-guarantee reasoning verifies threads individually with the help of assertions about other threads [38, 39]. Thread modular analysis relies on a similar technique. It verifies each thread individually using an abstraction of steps that may be performed by other threads [40–43].

Hybrid race detection systems have been developed that combine various techniques to overcome some of the limitations of these methods. Permission regions use static program instrumentation combined with dynamic analysis to detect races [44, 45]. Gradual permission regions use a similar program instrumentation along with model checking [25].

This work makes use of the happens-before relation for dynamic analysis of programs and uses model checking to ensure all schedules are considered in programs with mutual exclusion. A lot of different techniques create models of programs from program executions and use the models for verification. SATCheck observes the program execution to build a concrete behavior model of program execution and using a SAT solver, it tries to find other interesting behaviors [46]. Coverage driven testing uses program execution to create a model of the thread interleavings and shared memory accesses to identify unexplored thread interleavings [47, 48]. Regression testing tools for concurrent programs use changes in the program model to identify shared memory accesses that might be affected by the code changes and identifying thread interleavings that must be explored

المنارة للاستشارات
www.manaraa.com

to expose regression bugs [49, 50]. Dynamic symbolic execution is combined with unfolding of petri-nets to create minimal test-suites for testing multi-threaded programs [51, 52].

للاستشارات المنارة
www.manaraa.com

Chapter 9

Conclusion and Future Work

9.1 Conclusion and future work

This work presents a sound and complete technique for data race detection in task parallel programs using computation graphs. A dynamic improvement for the data race detection algorithm called on-the-fly analysis is also described. 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. The data race detection analysis is implemented for Java implementation of Habanero programming model using Java Pathfinder and evaluated on a host of benchmarks. The results are compared to JPF's precise race detector and gradual permission regions based extension. The results show that this technique reduces the time required for verification significantly. The results for data race detection using computation graphs are also compared to the on-the-fly analysis to demonstrate the performance gain it offers.

This work can be extended in the following ways:

- The data race detector based on computation graphs explores just one control flow path that is taken by the program execution based on the input. The listener can be extended to explore other control flow paths by using Symbolic Execution.
- The computation graphs can be created statically using program instrumentation and analyzed to gain performance improvements.

المنارة للاستشارات
www.manaraa.com

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37(1), pp. 55–69, 1996.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," *SIGPLAN Notices*, vol. 40(10), pp. 519–538, 2005.

[3] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the new adventures of old X10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, 2011, pp. 51–61.

[4] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," *ACM SIGOPS Operating Systems Review*, vol. 37(5), pp. 237–252, 2003.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (SIGPLAN Notices)*, vol. 37(5), pp. 234–245, 2002.

[6] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28(2), pp. 207–255, 2006.

[7] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," *SIGPLAN Notices*, vol. 41(6), pp. 308–319, 2006.

[8] J. W. Voung, R. Jhala, and S. Lerner, "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*, 2007, pp. 205–214.

[9] J.-D. Choi, A. Loginov, and V. Sarkar, "Static datarace analysis for multithreaded object-oriented programs," Technical Report RC22146, IBM Research, Tech. Rep., 2001.

المنارة للاستشارات

www.manaraa.com

[10] M. Vechev, E. Yahav, R. Raman, and V. Sarkar, "Automatic verification of determinism for structured parallel programs," in *Proceedings of the 17th International Conference on Static Analysis*, 2010, pp. 455–471.

[11] S. Kulikov, N. Shafiei, F. Van Breugel, and W. Visser, "Detecting data races with java pathfinder," 2010. [Online]. Available: http://nastaran.ca/files/race.pdf

[12] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp, "Implementing efficient dynamic formal verification methods for mpi programs," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2008, pp. 248–256.

[13] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 174–186.

[14] P. Anderson, B. Chase, and E. Mercer, "JPF verification of Habanero Java programs," *ACM SIGSOFT Software Engineering Notes*, vol. 39(1), pp. 1–7, 2014.

[15] M. Gligoric, P. C. Mehlitz, and D. Marinov, "X10X: Model checking a new programming language with an 'old' model checker," in *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 11–20.

[16] T. K. Zirkel, S. F. Siegel, and T. McClory, "Automated Verification of Chapel Programs using Model Checking and Symbolic Execution." *NASA Formal Methods*, vol. 7871, pp. 198–212, 2013.

[17] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," *SIGPLAN Notices*, vol. 44(6), pp. 121–133, 2009.

[18] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15(4), pp. 391–411, 1997.

[19] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1991, pp. 24–33.

[20] D. Schonberg, "On-the-fly detection of Access Anomalies," *SIGPLAN Notices*, vol. 24(7), pp. 285–297, 1989.

المنارة للاستشارات
www.manaraa.com

[21] M. Feng, “Efficient detection of determinacy races in Cilk programs,” in *In Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1997, pp. 1–11.

[22] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, “Efficient data race detection for async-finish parallelism,” in *International Conference on Runtime Verification*, 2010, pp. 368–383.

[23] ——, “Scalable and Precise dynamic datarace detection for structured parallelism,” *SIGPLAN Notices*, vol. 47(6), pp. 531–542, 2012.

[24] J. B. Dennis, G. R. Gao, and V. Sarkar, “Determinacy and repeatability of parallel program schemata,” in *Data-Flow Execution Models for Extreme Scale Computing*, 2012, pp. 1–9.

[25] E. Mercer, P. Anderson, N. Vrvilo, and V. Sarkar, “Model checking task parallel programs using gradual permissions,” in *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 535–540.

[26] A. Bouajjani and M. Emmi, “Analysis of recursively parallel programs,” *SIGPLAN Notices*, vol. 47(1), pp. 203–214, 2012.

[27] JGraphT, A Java Library focused on data structures and algorithms. [Online]. Available: http://jgrapht.org/

[28] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan, “Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs,” *SIGPLAN Notices*, vol. 37(5), pp. 258–269, 2002.

[29] T. Elmas, S. Qadeer, and S. Tasiran, “Goldilocks: Efficiently computing the happens-before relation using locksets,” in *Formal Approaches to Software Testing and Runtime Verification*, 2006, pp. 193–208.

[30] ——, “Goldilocks: A race and transaction-aware java runtime,” *SIGPLAN Notices*, vol. 42(6), pp. 245–255, 2007.

[31] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen, “Commutativity race detection,” *SIGPLAN Notices*, vol. 49(6), pp. 305–315, 2014.

[32] A. Nistor, D. Marinov, and J. Torrellas, “Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing,” in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 251–262.

المنارة للاستشارات
www.manaraa.com

[33] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21(7), pp. 558–565, 1978.

[34] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, "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*, 2009, pp. 13–22.

[35] V. Kahlon and C. Wang, "Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs," in *Computer Aided Verification*, 2010, pp. 434–449.

[36] B. P. Miller and J.-D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988, pp. 135–144.

[37] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in mpi programs," in *Formal Methods*, 2014, pp. 263–278.

[38] Q. Xu, W.-P. de Roever, and J. He, "The rely-guarantee method for verifying shared variable concurrent programs," *Formal Aspects of Computing*, vol. 9(2), pp. 149–174, 1997.

[39] C. Popeea and A. Rybalchenko, "Compositional termination proofs for multi-threaded programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2012, pp. 237–251.

[40] C. Flanagan and S. Qadeer, "Thread-modular model checking," in *Model Checking Software*, 2003, pp. 213–224.

[41] A. Malkis, A. Podelski, and A. Rybalchenko, "Precise thread-modular verification," in *Static Analysis*, 2007, pp. 218–232.

[42] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer, "Thread-modular abstraction refinement," in *International Conference on Computer Aided Verification*, 2003, pp. 262–274.

[43] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv, "Thread-modular shape analysis," *SIGPLAN Notices*, vol. 42(6), pp. 266–277, 2007.

[44] E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar, "Practical permissions for race-free parallelism," in *ECOOP–Object-Oriented Programming*, 2012, pp. 614–639.

المنارة للاستشارات

www.manaraa.com

[45] ——, “Permission regions for race-free parallelism,” in *Runtime Verification*, 2012, pp. 94–109.

[46] B. Demsky and P. Lam, “Satcheck: SAT-directed stateless model checking for SC and TSO,” in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 20–36.

[47] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, “Testing concurrent programs to achieve high synchronization coverage,” in *Proceedings of the International Symposium on Software Testing and Analysis*, 2012, pp. 210–220.

[48] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, “Maple: a coverage-driven testing tool for multithreaded programs,” *SIGPLAN Notices*, vol. 47(10), pp. 485–502, 2012.

[49] V. Terragni, S.-C. Cheung, and C. Zhang, “Recontest: Effective regression testing of concurrent programs,” *IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 246–256, 2015.

[50] T. Yu, W. Srisa-an, and G. Rothermel, “SimRT: an automated framework to support regression testing for data races,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 48–59.

[51] H. P. d. Leon, O. Saarikivi, K. Kahkonen, K. Heljanko, and J. Esparza, “Unfolding based minimal test suites for testing multithreaded programs,” in *15th International Conference on Application of Concurrency to System Design (ACSD)*, 2015, pp. 40–49.

[52] K. Kähkönen, O. Saarikivi, and K. Heljanko, “Unfolding based automated testing of multithreaded programs,” *Automated Software Engineering*, vol. 22(4), pp. 475–515, 2015.

المنارة للاستشارات
www.manaraa.com