Proof Paper 2 - Data Race Detection Algorithm

Radha Nakade
Department of Computer Science
Brigham Young University
Provo, Utah

January 11, 2016

Abstract

The Habanero Java (HJ) programming model is designed to provide several correctness guarantees such as determinism and serialization in the absence of data races. Model checking tools like Java Pathfinder (JPF) can be used to detect data races in HJ programs although as the program size increases, state space size grows exponentially and data race detection becomes expensive because of the state space explosion. We present a new method of data race detection in HJ programs with the help of computation graphs. A computation graph represents the execution of a program in the form of a directed acyclic graph. A computation graph can used to locate data races in programs by finding partially ordered set of tasks and the checking the memory locations accessed by these tasks.

1 Theorem

Given an HJ program with input ψ, Algorithm 3 reports a data race if and only if a data race exists in the program.

2 Translation of Theorem

For a given HJ program P with a fixed input, if the data race detection algorithm reports that the program is data race free, then no race can exist in any execution of the program. If the algorithm reports a race, then a data race definitely exists in some schedule of the program.

3 Introduction

The increasing use of multi-core processors is motivating the use of parallel programming. However, it is very difficult to write concurrent programs that are free from bugs. 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, deadlocks, etc. To make writing concurrent programs easier, Rice University developed the Habanero programming model [2]. It provides safety guarantees such as deterministic output and serialization for subsets of constructs provided in the programming model. These guarantees hold only in the absence of data-races. HJ-lib is a Java library implementation of the Habanero programming model.

VR-lib [1] is a verification runtime for HJ programs that facilitates the verification of HJ programs using JPF. VR-lib can be used along with JPF to build computation graphs of HJ programs. A Computation Graph (CG) is an acyclic directed graph that consists of a set of nodes, where each node represents a step consisting of some sequential execution of the program and a set of edges that represent the ordering of the steps. A CG stores the memory locations accessed and updated by each of the tasks.

4 Habanero Java Overview

The Habanero Java Programming model was built as an extension to the Java-based definition of the X10 language. It includes a set of powerful parallel programming constructs that can be used to create programs that are inherently safe. HJ puts particular emphasis on the usability and safety of parallel programming constructs.

4.1 HJ constructs for creating new tasks

Async and finish constructs are used to create and join tasks created by a parent process. The statement **async**(() $\rightarrow \langle$ **stmt1** $\rangle$) creates a new task to execute **stmt1** in parallel with its parent task. The Finish method is used to represent join in Habanero Java. The task executing **finish(()** $\rightarrow \langle$**stmt2**$\rangle$) has to wait for all the tasks running inside **stmt2** to finish before it can move on.

4.2 HJ Sample Program

Fig. 1 shows a sample program written in HJ. In this example, the main process starts two new tasks running in parallel with the process. The main process has to stop its execution at the end of finish block and wait for the children processes to complete their execution before the main process can proceed further. This results in a computation graph shown in Fig. 2.

5 Computation graphs

The execution of a task-parallel program can be represented in the form of a computation graph.

```
public class Example1{
        static int x = 0;
        public static void main(String[] args) {
                launchHabaneroApp(() -> {
                        finish(() -> {
                                async(() -> { //Task1
                                        int t = x;
                                        x = t+1;
                                });
                                async(() -> { //Task2
                                        int u = x;
                                        x = u+2;
                                });
                        });
                        System.out.println("Value of x = " + x);
                });
        }
}
```

Figure 1: Sample HJ Program

Definition 1. ***Computation Graph:*** *A Computation Graph* $G = \langle N, N_{tasks}, N_{finishscopes}, E, \delta \rangle$*, of an HJ program P with input* ψ *is a directed acyclic graph where*

- N *is a finite set of nodes*
- E *is a set of directed edges that represent ordering constraints.*
- $N_{finishscopes}$ *is set of finish nodes such that* $N_{finishscopes} \subseteq N$ *and each node represent the scope of various tasks in the computation graph*
- N_{tasks} *is set of task nodes such that* $N_{tasks} \subseteq N$ *and each node represents a step consisting of some sequential computation in the tasks*
- δ *is the function that maps* N_{tasks} *to the heap locations accessed by the corresponding tasks.*

Definition 2. ***Task Nodes:*** *Task nodes represent the various tasks that are spawned in the HJ program. The task nodes are used to store the memory locations accessed by the tasks in the HJ program along with the nature of operation (read or write) performed on that particular memory location.*

Definition 3. ***Finish Nodes:*** *Finish nodes are used to represent the scope of the tasks in which they can run.*

Definition 4. ***Mapping function*** δ***:*** *The mapping function delta maps the task nodes to the memory locations accessed by these tasks.*

$$\delta : N_{tasks} \longrightarrow MemLoc(R, W)$$

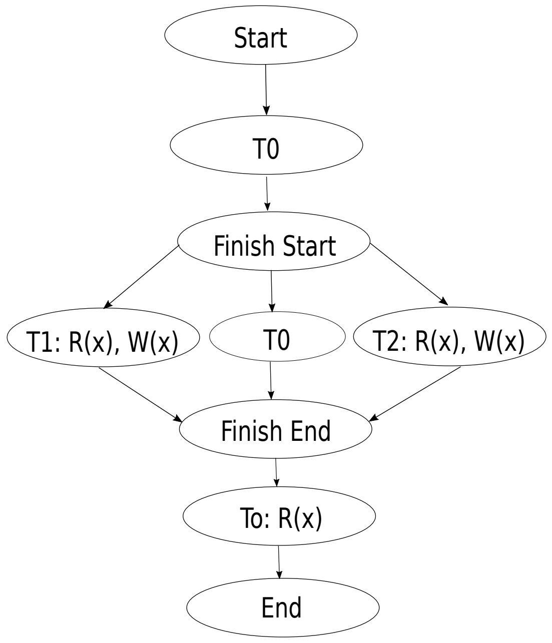

Figure 2: Computation Graph of the Sample program

5.1 Computation Graph Creation

Algorithm 1 is used to create the computation graph of an HJ program. Whenever a new object is created in the HJ program, the CreateNodes method is called and the object id and the task that created it are passed as arguments. The method checks if the newly created object belongs to Activity or FinishScope class. If the object belongs to a class other than Activity or FinishScope, then the method returns without making any changes to the computation graph.

5.2 Storing access info in task nodes

Whenever a heap location is accessed, a corresponding entry has to be made in the computation graph. Algorithm 2 is used to update the computation graphs with the heap locations that are accessed by the tasks in the HJ program.

Theorem 1. *Every HJ program with a given input ψ has a unique computation graph that corresponds to all possible executions.*

Proof. Consider an HJ program P with finish and async constructs. The imme-

Algorithm 1 Computation graph creation

```
function CreateNodes($O, T$)
    if O instanceof FinishScope then
        add a new node n of type $N_{finishscopes}$ to the computation graph
        if A thenn IEF node $n'$ is present in thread $T$
            add an edge from $n'$ to $n$
        else
            Add an edge from task node $O$ to $n$
        end if
    else if O instaceof Activity then
        add a new node $n$ of type $N_{tasks}$ to the computation graph
        if A thenn IEF node $n'$ is present in thread $T$
            add an edge from $n'$ to $n$
        else
            add a new node $n'$ of type $N_{finishscopes}$ to the computation graph
            Add an edge from task node $n'$ to $n$
        end if
    else
        Return
    end if
end function
```

Algorithm 2 Storing access info in task nodes

```
function StoreAccessInfo($M$, $Insn$, $N$)
    if Insn is Read then
        add M to the read list of $N$ in the computation graph
    else
        add M to the write list of $N$ in the computation graph
    end if
end function
```

diately enclosing finish for every statement in P is the same across all possible executions of P for a given input ψ. Also every statement in P belongs to the same task across all possible executions of P for input ψ. Hence, in every computation graph of P that corresponds to different executions of P for an input ψ, the finish scope of every async statement is same in P. Hence, there is a unique computation for every HJ program with finish and async constructs for a given input. □

6 Data Race Detection

Data races occur in parallel programs when two or more processes executing in parallel access a shared memory location and at least one of them is a write. A data race means that the value of the memory location will be dependent on the order of execution of the instructions that accessed it and it is hard to determine the exact value of that memory location at the end of the execution of that task.

Definition 5. ***Data Race:*** *Two accesses to a shared memory location by two different tasks result in a data race if:*

- *At least one access is a write.*
- *The accesses are partially ordered. (i.e. There is no total ordering between the two accesses).*

Algorithm 3 Data race detection in a computation graph

```
function DetectRace(Graph G)
    for all finish_nodes $n \in G$ do
        task_nodes := Search(G, n)
        validate(task_nodes)
    end for
end function
function validate(task_nodes)
    for all $n \in task\_nodes$ do
        for all $n' \in task\_nodes \wedge (n \neq n') \wedge \neg(n \rightarrow n')$ do
            if $H(n) \cap W(n') \neq \phi$ then
                Report Data Race and Exit
            end if
        end for
    end for
end function
```

The data race detection algorithm takes the graph and for every pair of finish-start and finish-end nodes, it creates a task list consisting all task nodes present in this finish-scope. All these task nodes execute in parallel if there is no path between these nodes. The global memory accesses by these tasks is then checked and if conflicting memory-accesses are observed, a data race is reported.

Definition 6. ***Soundness:*** *A data race detection algorithm is sound if it does not miss any data races in the programs. If the sound algorithm declares a program is data race free, no race can exist in execution of the program for the given input. 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 7. ***Completeness*** *A data race detection algorithm is complete if it does not report any false positives; i.e., if the algorithm reports a data race, then the data race really exists. It 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.*

Lemma 1. *Given an HJ program with an input ψ, Algorithm 3 is complete.*

Proof. The computation graph G is a directed acyclic graph. The transitive closure of the graph gives the reachability relation of all the dag. It is a strict partial order over the nodes of the graph. Since the task nodes belonging to a finish scope are not reachable from each other, these nodes are partially ordered. Hence, by definition of data race, Algorithm 3 is complete. $\square$

Lemma 2. *Given an HJ program with an input ψ, Algorithm 3 is sound.*

Proof. Suppose Algorithm 3 is not sound. i.e. there exists a data race in the HJ program that is not reported by algorithm 1. Therefore, there are at least two tasks t_1 and t_2 executing in parallel that have conflicting accesses to a shared memory location L. Therefore, there exists a computation graph G$'$ that contains two task nodes corresponding to tasks t_1 and t_2 that belong to a common finish scope and access memory location L.

Graphs G and G$'$ represent the same HJ program but have different structures. This contradicts theorem 1 which states that every HJ program has a unique computation graph. Therefore, our assumption is wrong. Hence, Algorithm 1 is sound for a given input. $\square$

Theorem 2. *Given an HJ program with an input ψ, Algorithm 3 reports a data race if and only if a data race exists in the program.*

Proof. From lemma 1 we see that the algorithm reports a race if a data race exists in the program. From lemma 2 it follows that if the HJ program has a data race, algorithm 3 will definitely report it.

$\square$