Verification of Habanero Java Programs using Computation Graphs

Radha Nakade
Brigham Young University
Provo, Utah
radha.nakade@gmail.com

Eric Mercer
Brigham Young University
Provo, Utah
egm@cs.byu.edu

ABSTRACT

The Habanero Java (HJ) programming model is designed to provide several correctness guarantees such as deadlock freedom and determinism in the absence of data races. Model checking tools like Java Pathfinder (JPF) can be used to detect data races in HJ programs. But, as the program size increases, 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. For structurally deterministic programs, analyzing the graph on a single schedule is enough for data race detection.

1. INTRODUCTION

The increasing use of multicore 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 Habanero Java Programming model [3]. It provides safety guarantees such as deadlock freedom, deterministic output and serialization for subsets of constructs provided in the programming model. These guarantees hold only in the absence of data-races. The Habanero Java library (HJ-Lib) [8] is a Java 8 library implementation of the Habanero Java programming model.

VR-lib [2], a verification runtime for HJ programs was built at Brigham Young University. VR-lib 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 operators. It also correctly reflects the control flow structure of the program.

The CGRaceDetector listener presented in this work monitors the various object creations and destructions, instruction executions etc to build a computation graph for the HJ program under a single schedule. It later analyzes this graph to verify any data access violations to report data races. For structurally deterministic programs, verifying the HJ program under a single schedule is enough to detect data races.

Section 2 of this paper presents an overview of the Habanero Java programming model and gives a brief description for the various parallel constructs of HJ language. Section 3 describes the computation graphs and its various elements. It also gives the implementation details of computation graph builder and analyzer for HJ programs created with the help of JPF. Section 4 describes the preliminary results of the CGRaceDetector on some HJ micro-benchmarks. Section 5 concludes and outlines the ways to extend this work.

2. HABANERO JAVA OVERVIEW

The Habanero Java Programming model was built as an extension to the Java-based definition of the X10 language. The Habanero Java Library (HJ-lib) is a Java 8 library implementation of the Habanero Java Programming model. It includes a set of powerful parallel programming constructs that can be used to create programs that are inherently safe. HJ-Lib puts particular emphasis on the usability and safety of parallel programming constructs. For example, no HJ program written using async, finish, isolated and phaser constructs can create a logical deadlock cycle. HJ-Lib creates standard Java class files that can be run on any Java 8 JVM.

2.1 HJ constructs

HJ consists of a wide range of constructs for parallel programming.

- **Task Spawn and Join:** Async and finish constructs are used to create and join tasks created by a parent process. The statement async($() \rightarrow \langle stmt \rangle$) creates a new task that can logically execute in parallel with its parent task. The Finish method is used to represent join in Habanero Java. The task executing finish($() \rightarrow \langle stmt \rangle$) has to wait for all the tasks running inside stmt to finish before it can move on.
- **Loop Parallelism:** The forall and forasync constructs in HJ are used for loop parallelism. An implicit finish is included at the end of forall iterations whereas forasync iterations do not have an implicit finish.
- **Coordination Constructs:** There are often dependencies among parallel tasks. To coordinate the execution of the parallel tasks, HJ provides some constructs such as isolated, futures, data-driven futures and phasers.
  - **Isolated:** Most of the concurrently running processes have the need to synchronize the access to the shared variables. The isolated statements allow only one process at a time to access referenced shared variables. Isolated statements create performance bottlenecks in moderate to high contention systems. HJ also provides object-based isolation which provides better performance.
  - **Futures:** HJ supports returning values from a newly created child task to the parent task with the help of

```
public class Example1{
    static int x = 0;
    public static void main(String[] args) {
        launchHabaneroApp(() -> {
            finish(() -> {
                async(() -> { //Thread1
                    isolated(() -> {
                        x++;   //Isolated block s
                    });
                });
                async(() -> { //Thread2
                    isolated(() -> {
                        x++;   //Isolated block s'
                    });
                });
            });
        });
    }
}
```

Figure 1: Sample HJ Program

futures. The statement HjFuture⟨T⟩ f = future ⟨T⟩ (() → ⟨expr⟩) creates a new child task which executes expr and the result of this execution can be obtained by the parent task by calling f.get() method.

- **Data-driven futures:** DDFs are an extension to futures. Any async can register on a DDF as a consumer causing the execution of the async to be delayed until a value becomes available in the DDF. The exact syntax for an async waiting on a DDF is as follows: asyncAwait(ddf1, ... , ddfN, () → stmt). An async waiting on a chain of DDFs can only begin executing after a put() has been invoked on all the DDFs.
- **Phasers** - Phasers help in point-to-point synchronization. Each task has the option of registering with a phaser in signal-only/wait-only mode for producer/-consumer synchronization or signal-wait mode for barrier synchronization. A task may be registered on multiple phasers, and a phaser may have multiple tasks registered on it. Phasers ensure deadlock freedom when programmers use only the next statements in their programs. In programs where tasks are involved with multiple point-to-point coordination, explicit use of doWait() and doSignal() on multiple phasers might be required.

2.2 HJ Sample Program

Fig. 1 shows a sample program written in HJ. In this example, the main process starts two new processes running in parallel with the process. The main process has to stop its execution at the end of finish block and wait for the child processes to complete their execution before the main process can proceed further. Both the newly spawned processes have a co-ordination construct 'isolated' that creates nodes s and s' in the computation graph. There is a serialization edge between s and s' that shows the ordering of the events in this execution. This results in a computation graph shown in Fig. 2.

3. COMPUTATION GRAPHS

The execution of a task-parallel program can be represented in the form of a computation graph. A computation graph of a program is a directed acyclic graph (DAG) structure that represents the sequence of execution of tasks in the parallel program. A computation graph can be represented as G = ⟨V, E⟩ where

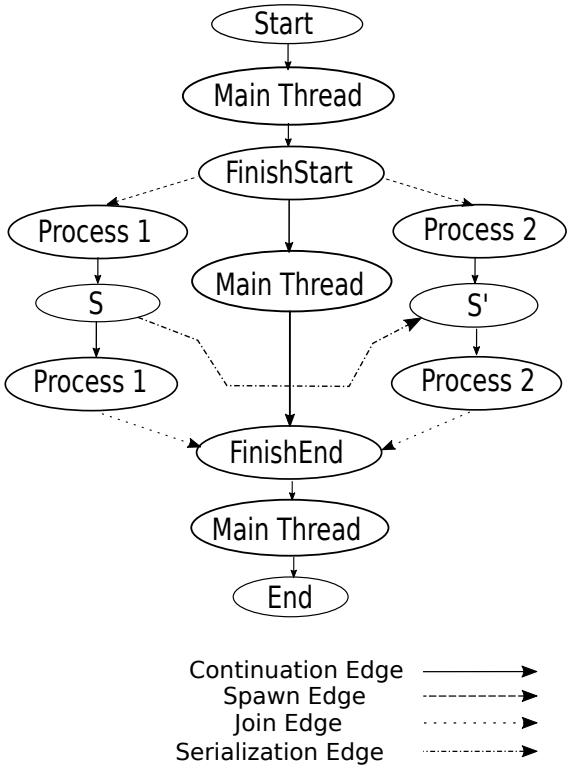

Figure 2: Computation Graph of the Sample program

- V is a set of nodes such that each node represents a step consisting of an arbitrary sequential computation and
- E is a set of directed edges that represent ordering constraints. The various types of edges in a computation graph are:
  - **Spawn edges:** They connect steps in parent tasks to steps in child async tasks. When an async is created, a spawn edge is inserted between the step that ends with the async in the parent task and the step that starts the async body in the new child task.
  - **Join edges:** They connect steps in descendant tasks to steps in the tasks containing their Immediately Enclosing Finish (IEF) instances. When an async terminates, a join edge is inserted from the last step in the async to the step that follows the IEF operation in the task containing the IEF operation.
  - **Continue edges:** They capture sequencing of steps within a task - all steps within the same task are connected by a chain of continue edges.
  - **Serialization edges:** They connect two isolated nodes S and S' where nodes S and S' are interfering. Two isolated nodes do not interfere only if they have a total ordering in the CG.

Computation graphs provide a visual feedback of the execution of HJ programs. Computation graphs can be used not just to find concurrency bugs in parallel programs but also to optimize the code. In this work, we concentrate only on data race detection in HJ programs with the help of computation graphs.

Implementation Details: The CG builder is implemented using JPF. It uses the VR-lib, specifically designed to run HJ programs using JPF. The HJ program is compiled using the VR and the class files are analyzed using JPF. JPF creates thread choice generators to systematically explore the state-space of the programs. We choose one set of thread interleavings to build a CG for that execution. JPF tracks references to all the variables in the program. It also computes the aliasing information during runtime. These memory references are stored in the CG. The CGs are stored in a DAG data structure.

Data races can be detected in a CG when two parallel nodes in the graph access a memory location and at least one of the operations tries to modify it. A topological traversal of the graph gives the order of execution of the various tasks. All the nodes that occur between a pair of Finish-start and Finish-end nodes execute in parallel. The global memory accesses by these processes is checked and if conflicting memory-accesses are observed, then a data race is reported.

4. RESULTS

We verified some of the HJ microbenchmarks that make use of only the basic parallel constructs such as async and finish using the CGRaceDetector listener. The CGRaceDetector is able to build computation graphs of the HJ programs by exploring very few states. We compared the output of CGRaceDetector to the output of PreciseRaceDetector and found that CGRaceDetctor was able to correctly identify races in all programs. These microbenchmarks are variations of a linear search algorithm. The first test finds the count of occurrences of a search string in a given text string. The second test confirms the existence of search string in the given text string. The third test returns the index of occurrence of the search string. In case of multiple occurrences, the output becomes non-deterministic. The fourth test also confirms the existence of the search string in the given text. However, as soon as the search text is found, no more processes are spawned to search the text and the program is terminated. Similarly, in the fifth test, as soon as a process returns the index of occurrence of search text, the program terminates. The results are presented in Table I. The sizes of the programs are indicated by the SLOC column and Tasks columns represents the number of tasks created in every program. The results of CGRaceDetector and Precise Race Detector are compared. The Precise Race Detector systematically explores the entire state space of the program. The CGRaceDetector just uses one thread interleaving to detect data races. Hence, the time required by CGRaceDetector is considerably smaller than the time required by Precise Race Detector to execute.

5. RELATED WORK

JPF has been extensively used to verify task-parallel programs written in various languages. There is an extension in JPF for model-checking X10 programs [7]. Similarly, an extension of JPF has been developed to verify Chapel programs [15]. The work presented here makes use of the Verification Runtime [1] and VR-lib [2] developed at Brigham Young University. The VR-lib facilitates the verification of Habanero Java Programs using JPF.

Many tools have been developed for verifying the correctness of task-parallel programs. Dynamic data race detection tools monitor the shared-memory accesses and verify that consistent locking behavior is used [12], [13], [4]. Static race detection tools [14], [5], [6] use program instrumentation to identify program fragments that execute in parallel and verify that such fragments perform independent memory accesses.

Graph based approaches have also been used in the past to detect data races in parallel programs. Miller and Choi implemented an integrated debugging system for parallel programs in [11]. They created dynamic program dependence graphs that show the causal relations between program events. The dynamic program dependence graph is created by observing a trace of the parallel program execution. They also showed how the dynamic program dependence graph can be used to detect data-races in parallel programs. A method to perform static data race detection in concurrent C programs was developed by Kahlon et al. This method [9] involved creating a precise context-sensitive concurrent control flow graph to identify the shared variables and lock pointers, compute the initial database of race warnings and then prune away the spurious messages using may-happen-in-parallel (MHP) analysis. Kahlon and Wang proposed a concept of Universal Causality Graphs (UCG) in [10]. UCGs encode the set of all feasible interleavings that a given correctness property may violate. UCGs provide a unified happens-before model by capturing causality constraints imposed by the property at hand as well as scheduling constraints imposed by synchronization primitives as causality constraints.

6. CONCLUSION AND FUTURE WORK

In this work, we presented a new way of detecting data-races in HJ programs with the help of computation graphs. The CGRaceDetector explores fewer states compared to the number of states explored by JPF's PreciseRaceDetector. This work can be further extended in the following ways:

1. The CGRaceDetector 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 as well by using Symbolic Execution.
2. The CGRaceDetector listener can be extended to include co-ordination constructs such as phasers and Data Driven Futures.
3. To detect benign data-races, a new listener can be implemented that creates shared access choice generators for only the variables that were a part of the dara race. If the program produces deterministic results, the race condition is benign.

7. REFERENCES

[1] P. Anderson, B. Chase, and E. Mercer. JPF Verification of Habanero Java Programs. *SIGSOFT Softw. Eng. Notes*, 39(1):1–7, Feb. 2014.

[2] P. Anderson, N. Vrvilo, E. Mercer, and V. Sarkar. JPF Verification of Habanero Java Programs Using Gradual Type Permission Regions. *SIGSOFT Softw. Eng. Notes*, 40(1):1–5, Feb. 2015.

[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*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[4] 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 Not.*, 37(5):258–269, May 2002.

[5] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

Table 1: Verification of HJ Micro-benchmarks using CGRaceDetector

		CGRaceDetector			Precise Race Detector			
Test Case Name	SLOC	Tasks	States	Time	Error Info	States	Time	Error Info
Search Count	50	4	195	0:00:01	No Race	145139	0:00:45	No Race
Existence of an occurrence	45	4	174	0:00:01	Detected Race	50197	0:00:15	Detected Race
Index of occurrence	38	4	197	0:00:01	Detected Race	68806	0:00:29	Detected Race
Existence of occurrence with no task creation after instance is found	45	2	117	0:00:00	Detected Race	296	0:00:00	Detected Race
Search Index With No task creation after Instance is Found	48	2	119	0:00:00	Detected Race	326	0:00:00	Detected Race

[6] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

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

[8] S. Imam and V. Sarkar. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 75–86, New York, NY, USA, 2014. ACM.

[9] 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*, pages 13–22. ACM, 2009.

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

[11] B. P. Miller and J.-D. Choi. *A mechanism for efficient debugging of parallel programs*, volume 23. ACM, 1988.

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

[13] 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)*, 15(4):391–411, 1997.

[14] M. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic verification of determinism for structured parallel programs. In *Static Analysis*, pages 455–471. Springer, 2011.

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