Thesis Proposal:
Verification of Task Parallel Programs

by

Radha Nakade

A thesis proposal, submitted to the faculty of Brigham Young University in partial fulfillment of the requirements for the degree of Master of Science.

Department of Computer Science

Brigham Young University

March 9, 2015

Abstract

Parallel programming is being used increasingly in large-scale applications. Task parallelism is a branch of parallel programming in which different instructions are run on different sets of data simultaneously on the available processor cores. This gives rise to different memory access patterns. When two or more processes access a memory location such that at least one of the accesses is a 'write', a data-race is created in the program. Data-races cause the output of the program to be non-deterministic. A computation graph of a task parallel program can be used to represents the execution of the program in the form of a directed acyclic graph. In this research, we propose a method for detecting data-races with the help of a computation graph of the program. We also discuss ways to test the effectiveness of this method.

1 Introduction

Until recently, the speed of the processors was expected to increase rapidly over time with sustained technology advances, and the motivation for parallel computing was low. But now, clock frequencies for individual processors are no longer increasing. The reason for that being the power consumed by a processor using current device technologies varies as the cube of the frequency. Processor chips in laptop and personal computers are typically limited to consume at most 75 Watts because a larger power consumption would lead to chips overheating and malfunctioning, given the limited air-cooling capacity of these computers. This power limitation has been referred to as the "Power Wall". The Power Wall is even more critical in smart-phones and hand held devices because larger power consumption leads to a shorter battery life. Therefore, clock frequency can no longer be increased. When multiple cores are used in parallel, the speed of computation is increased but not the power consumption. This is the main motivation behind parallel programming.

However, the introduction of concurrency leads to non deterministic behavior of programs. When programs execute different instructions simultaneously, different thread-schedules and memory accesses patterns are observed. If two or more processes executing in parallel access a common memory location and atleast one of them is a write, then a data race occurs. Data-race means that the value of the memory location will be dependent on the order 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. Data-race occur under only certain thread-interleavings. To reproduce a data-race, the same thread-schedule has to be followed. Hence, data-races are quite hard to detect. Another problem that can arise when two or more tasks execute in parallel is deadlock. When a task is waiting to acquire a resource held by another task and the other task is waiting to acquire a resource held by the first task, then both tasks don't make any progress. This results in the execution of the program getting stalled. Many tools have been developed to detect deadlocks and data races in parallel programs. These tools assist the developers of concurrent programs to detect erroneous behavior of their programs and rectify their implementations. However, in large systems it takes a great deal of effort to locate and rectify such errors. Therefore, the developers are trying to develop parallel programming languages that ensure safety against concurrency related errors.

The research proposed here discusses a new way of verification of task parallel programming languages with the help of computation graphs. A computation graph of a program represents an execution of the program under certain thread-interleavings. A 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 should store the memory locations accessed and updated by each of the operators. It should also correctly reflect the control flow structure of the program. These properties are necessary for verification algorithms to correctly identify the errors in the programming constructs. Different computation graph structures are created for different inputs of the same program because of the control flow of the program. If all these computation graphs can satisfy the safety properties, then only we can claim that the program is safe to execute with any input. To enumerate all the computation graph structures of the program, we have to know all the possible inputs that can create these different structures. One way to do this is using concolic execution. Concolic execution is a hybrid verification technique that performs symbolic execution (treating program variables as symbols) with concrete execution (testing on particular inputs).

2 Thesis Statement

A computation graph is a suitable common representation of the execution of any task parallel program, and 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 and such an exhaustive enumeration is enough for general verification.

3 Related Work

Different parallel programming models have been created with different properties such as task interactions, task granularity, etc. Some examples of these models are message passing, data parallelism, task parallelism. Message passing [14] programs create multiple tasks with each encapsulating some local data. Data is shared between the tasks by sending messages from one task to another. MPICH is a widely used implementation of the MPI protocol. Data parallelism refers to application of same instruction to multiple elements of data. Task parallelism is achieved by executing different instructions concurrently on multiple sets of data.

In this research we are mainly going to focus on task parallel programs. Various task parallel languages have been developed such as Habanero Java, Cilk, X10, Chapel, OpenMP 3.0 etc. Habanero Java [7] is a task parallel programming language developed at the Rice University. It was developed as an extension to X10 with particular emphasis on safety properties of parallel constructs. The HJ compiler generates standard Java class files that can run on any JVM. The HJ runtime is responsible for orchestrating the creation, execution and termination of HJ tasks. A Java 8 implementation for this language, known as HJLib [17] has also been created. This library makes extensive use of lambda expressions and can run on any Java 8 JVM. HJ programs provide various safety guarantees if the parallel programming constructs are used correctly. Verifying HJ programs using tools such as Java Path Finder (JPF) can be time and memory consuming because of the numerous JPF state expansions. Hence, an HJ verification runtime (VR) [1] was developed at Brigham Young University to use JPF for verifying HJ programs. This runtime provides a lightweight alternative to verifying HJ programs using JPF.

X10 [9] was developed at IBM as a part of the IBM PERCS project (Productive Easy-to-use Reliable Computer Systems). X10 is a type-safe, parallel, distributed object oriented language with support for high performance computation over distributed multi-dimensional arrays. Gligoric et al. extended the model checker JPF in [15] to verify X10 programs and detect concurrency related bugs. Chapel programming language [8] was developed as a part of DARPA's High Productivity Computing Systems (HPCS) program. Chapel provides higher-level abstractions for parallelism using anonymous threads that are implemented by compiler and runtime system. Chapel programs are subject to concurrency problems such as deadlocks, race conditions etc. To verify the correctness of chapel programs, Zirkel et al. developed a tool for model checking and symbolic execution of chapel programs [28]. OpenMP [10] is a set of compiler directives and callable runtime library routines for Fortran and C to express shared-memory parallelism. Any sequential program written in C or fortran can be easily parallelized using OpenMP. Another C-based runtime system for multi-threaded parallel programming is Cilk [**?**]. A Cilk program consists of procedures that can be broken down into a sequence of threads. The performance of Cilk programs can be modeled accurately. This provides the developers a way to improve performance of their programs.

Model checking suffers from an inherent shortcoming. The exponential growth in the state space of the program being verified makes model checking unsuitable for large programs. A lot of methods are being developed such as partial order reductions that help to reduce the state space that needs to be explored for finding bugs such as data races which can be detected only under certain thread interleavings. Also, some errors are dependent on the control flow structure of the program. These errors are detected only if a given input takes that particular branch of the branch on which the error exists. To detect such errors we have test on all the branches on the program. Concolic execution provides a way of detecting errors on all possible branches of the program. Concolic execution automates test input generation by combining the concrete and symbolic execution of the code under test. Concolic execution couples both concrete and symbolic execution by running both of them simultaneously such that each gets feedback from the other.

Sen and Agha developed a concolic execution tool called CUTE in [26] for testing programs written in C and

Java. The tool consists of two modules: an instrumentation module and library to perform symbolic execution. The instrumentation module inserts code in the program to call the library at runtime for performing symbolic execution. Another concolic execution tool for Java programs was developed by Jayaraman et al. in [18]. This tool is called jFUZZ. It is a concolic white box fuzzer built on top of Java Path Finder(JPF). Starting from a seed input, jFUZZ automatically generates inputs that explore new program paths. Sen et al. developed a tool called DART (Directed Automated Random Testing) to automatically test softwares [16] . It combines three techniques: automated extraction of the interface of a program with its external environment using static source-code parsing; automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in; and dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths. DART has been implemented to test programs written in C.

Task parallel languages achieve parallelism by distributing execution processes across different parallel computing nodes. A formal model can be used to describe the properties of task parallel programming languages.

Semantics of parallel programming languages: Creating semantic models of programming languages helps to reason about the properties and performance of the various constructs of programming languages. This is an important step in the verification process of programming languages.

Emmi and Boujjani introduced an interleaving free model of isolated hierarchical parallel computations for expressing general parallel programming languages [4]. They formalized a system for measuring the complexity of deciding state reachability for finite-data recursive programs. Another way of creating formal models of programming languages is through Redex [21]. Redex is an executable domain-specific language for mechanizing semantic models developed by PLT. These models can be used to state theorems about the models and prove them. Redex is used by semantics engineers to formulate the syntax and semantics of the model, create test suites, run randomized testing and use graphical tools for visualizing examples etc.

Formal definitions of various properties of programming languages are also very important for the verification process.

Formalism of properties of Parallel Programming languages:

Scott and Lu have proposed various history-based definitions of determinism in [22]. They have discussed the comparative advantages of these defined properties. They have also discussed the containment relationships for these properties.

Dennis, Gao and Sarkar presented precise definitions of the two related properties of program schemata determinacy and repeatability in [11]. A key advantage of providing definitions for schemata rather than concrete programs is that it simplifies the task for programmers and tools to check these properties. The definitions of these properties are provided for schemata arising from data flow programs and task-parallel programs, thereby also establishing new relationships between the two models.

Race conditions occur in shared-memory parallel programs when accesses to shared-memory are not synchronized. Netzer and Miller formalized the definitions of race conditions occurring in shared-memory parallel programs [23]. The race conditions are divided broadly into two categories - general races that cause deterministic programs to fail in execution and data races that appear in non-deterministic programs.

Banerjee et al. developed a rigorous mathematical framework that can be used to study the trade-off between the amount of access history kept and the kinds of data races that can be detected [2]. Using this framework, they developed some algorithms for data race detection under different conditions.

Bocchino et al. developed a region-based type and effect system for expressing important patterns of deterministic parallelism in imperative, object-oriented programs [3]. This system simplifies parallel programming by guaranteeing deterministic semantics with modular, compile time type checking.

Kahlon and Wang proposed a concept of Universal Causality Graphs (UCG) in [20]. 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.

Using these formal definitions of various properties, various tools were developed for data-race detection, deadlock detection, checking determinism etc.

Checking Determinism:

In a parallel program, the threads of the parallel program can be interleaved non-deterministically during execution. Different thread interleavings result in different outputs for the same program input. Some of the results produced by such interleavings can be correct while others are wrong. Parallel programs should always produce the correct result irrespective of the thread interleavings that occur during program execution.

Burnim and Sen created an assertion framework that can be used to specify pairs of program state that can arise due to non-deterministic thread inter-leavings[5]. Such pairs of program state result in a deterministic result in spite of the different parallel schedules. They created a Java library that can be used to specify these assertions. They also created an algorithm called Determin [6] that can dynamically infer likely deterministic specification when provided with a set of inputs and schedules

Insta-check [24] is another technique for checking external determinism during testing of parallel programs. It checks whether different runs of a parallel program with same input produce different outputs. This is done by computing a 64-bit hash of the memory state during program run. If two program runs with same input produce different hashes, then insta-check reports that the program is non-deterministic.

Vechev et al. developed a static analysis technique for automatic verification of determinism in parallel programs [27]. The analysis is done in two phases. First phase identifies parts of the parallel program that run in parallel. Each part is sequentially analyzed by assuming that all memory locations accessed by the task are independent from locations accessed by other tasks that are running in parallel. In the second phase, the analysis checks whether this independence assumption holds i.e. all memory accesses are independent.

The main cause of non-determinism in parallel programs is data races and deadlocks that arise during different schedules of the program. To prove program correctness of parallel programs, it is important to detect all data races and deadlocks.

Data Race and Deadlock Detection:

Data races occur in parallel programs when two or more threads access a memory location and at least one of the accesses is a write. It is very difficult to detect data races in concurrent programs. Deadlocks cause the programs to stall. A number of researchers have worked on data race and deadlock detection.

Savage et al. developed a tool called Eraser [25] to dynamically detect data races in multi-threaded programs. Eraser uses binary rewriting techniques to monitor every shared-memory reference and verify that consistent locking behavior is used.

A method to perform static data race detection in concurrent C programs was developed by Kahlon et al. This method [19] involved creating a precise context-sensitive concurrent control flow graph. Using this graph, identify the shared variables and lock pointers, compute on initial database of race warnings and then prune away the spurious messages using may-happen-in-parallel (MHP) analysis.

Flanagan and Freund developed a precise data race detection tool called FastTrack [13]. It uses an adaptive lightweight representation for the happens-before relation that reduces both time and space overheads.

Engler and Aashcraft developed a static tool called RacerX for detection of deadlocks and race conditions [12]. The tool is specifically designed for checking large multi-threaded systems. It has been applied to Linux, FreeBSD etc. for detecting concurrency related errors in these complex systems.

4 Project Description

The research described here proposes a new technique for verification of task parallel programs with the help of computation graphs. It uses Habanero Java as an exemplar of task parallel languages. Habanero Java language is

a task parallel programming language built as an extension to X10 language. It includes a set of powerful parallel programming constructs that can be used to create programs that are inherently safe. HJ programs can be run on any JVM including Java 8 JVM. The Habanero Java language 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.

4.1 Background Information

4.1.1 HJ constructs

Habanero Java consists of a wide range of constructs for parallel programming.

1. **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.

2. **Loop Parallelism** - The forall and foreach constructs in HJ are used for loop parallelism. An implicit finish is included at the end of forall iterations whereas foreach iterations do not have an implicit finish.

3. **Coordination Constructs** - There are often dependencies among parallel tasks. To coordinate the execution of the parallel tasks, Habanero Java provides some constructs such as isolated, futures, data-driven futures and phasers.

   (a) **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.

   (b) **Futures** - HJ supports returning values from a newly created child task to the parent task with the help of futures. The statement HjFuture$\langle T \rangle$ f = future $\langle T \rangle$ ($() \rightarrow \langle expr \rangle$) 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.

   (c) **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, $() \rightarrow$ stmt). An async waiting on a chain of DDFs can only begin executing after a put() has been invoked on all the DDFs.

   (d) **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.

4.1.2 Computation Graphs

The execution of an 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 = $\langle V, E \rangle$ where

- V represents a set of nodes such that each node represents a step consisting of an arbitrary sequential computation and

- E represents a set of directed edges that represent ordering constraints. The various types of edges in a computation graph are:

    1. **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.
    2. **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.
    3. **Continue edges** - They capture sequencing of steps within a task - all steps within the same task are connected by a chain of continue edges.
    4. **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.

4.1.3 HJ example with its computation graph representation

Fig.1 shows an example program written in Habanero Java. 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. This results in a computation graph shown in fig. 2.

Figure 1: Example HJ Program

```
Public class Example1{
        Public static void Main(String[] args)
        {
                finish(() -> {
                        async(() -> {
                                //Thread1
                        });
                        async(() -> {
                                //Thread2
                        });
                });
        }
}
```

4.1.4 Properties of Task Parallel Programs

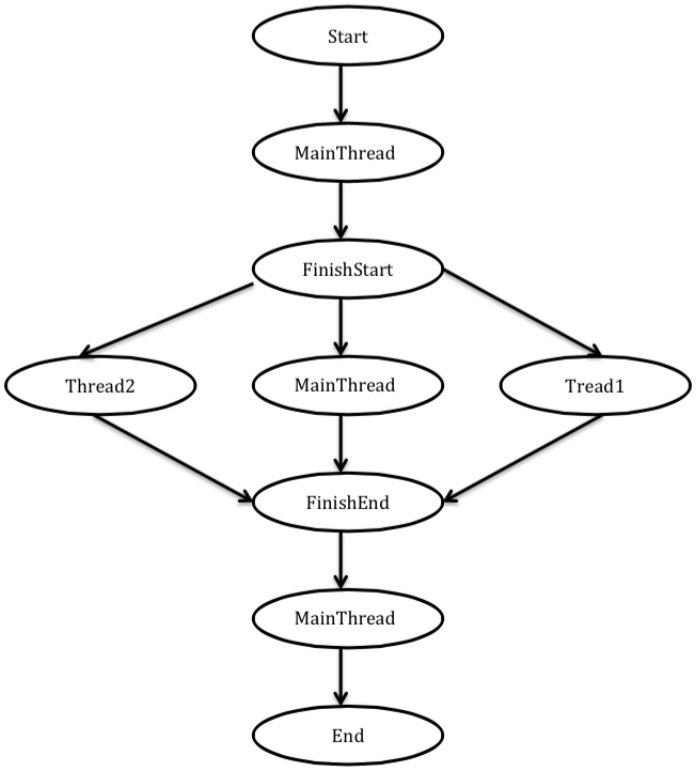
The following properties of task parallel programs can be verified using computation graphs.

1. **Data Race:** Data races occur in parallel programs when two or more tasks try to access shared variables such that at least one of the accesses is a 'write'. Data races cause the output of the program to become non-deterministic.

2. **Determinism:** Determinism refers to obtaining the same result from a parallel program for a given input. There are different definitions that determine the degree of determinism in parallel programs.

    (a) **Internal (or structural) determinism:**
    This type of determinism requires the parallel program to not only produce the same output for a given input, but also to produce a unique computation graph for a given input. A program is internally deterministic only if it is free of data-races.

Figure 2: Computation Graph of the example program



(b) **External (or functional) determinism:**
External determinism requires the parallel program to produce the same output when run on the same input. Program's executions for a given input may be different, but they ultimately produce the same output. In the presence of data-races, a program is externally deterministic if every pair of concurrent operators commute.

4.2 Implementation details

An HJ computation graph can be built using a tool called HJ-Viz. HJ-Viz processes the event logs produced by the HJLib runtime and generates a dot representation of the computation graph. The dot file can later be processed and displayed in the user's web browser using the HJ-Viz server. The drawback of this tool is that it does not store the computation graph in a logical data structure that can be traversed and analyzed to verify the properties of the program. Another drawback is that it does not save any memory references to objects. Hence, we need to create a tool that can build a computation graph of an HJ program during runtime and store the information in a logical data-structure that is easy to traverse and analyze.

The modified computation graph builder is implemented using Java Path Finder(JPF). It uses the Verification Runtime, 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 use one choice of thread interleavings at a time to observe the execution and build a computation graph for this execution. The VM listeners in JPF are extended to track the thread creations, executions, joins etc. Memory references are also registered in the computation graphs. The computation graphs are stored in DAG data structure. We are going to use jgrapht library that provides an implementation of this data structure.

Data races can be detected in a computation graph when two parallel nodes in the graph access a memory location and atleast 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 have to be checked and if conflicting memory-accesses are observed, then a data race should be reported.

For checking functional determinism of the program in the presence of data-races, JPF schedules on the memory location access that is part of the data race. If all the executions of the program produce the same result, then the program is functionally deterministic.

In programs with isolated regions (CG containing serialization edges), JPF creates schedules such that all possible orderings between the interfering isolated nodes is considered. The rest of the process for detecting data-races and checking determinism is carried out on all the obtained computation graphs.

5 Validation

To test the correctness of this approach, we have selected tests for which the properties are already known. The following table gives a summary of the tests and their properties.

Test Case	Data Race Free	Structurally deterministic	Functionally deterministic
Search Count	Y	Y	Y
Search	N	Y	Y
Search Index	N	Y	N
Search Index With No TaskCreation after Instance is Found	N	N	N

6 Thesis Schedule

The following schedule is an outline of submission dates for my Master's Thesis:

Submission to Advisor: February 8, 2014

Submission to Committee Members: February 22, 2014

Master's Thesis Defense: April 4, 2014

References

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

This paper introduces a verification runtime for Habanero Java programs. It extends Java Path Finder (JPF), an existing model checker for java to verify HJ programs.

[2] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 69–78. ACM, 2006.

This paper discusses a mathematical framework that can be used to study the tradeoff between the amount of access history kept and the kinds of data races that can be detected. It describes the necessary and sufficient conditions for two threads to have a data race. It also describes some algorithms to detect races in different conditions.

[3] Robert L Bocchino Jr, Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. *ACM Sigplan Notices*, 44(10):97–116, 2009.

This paper proposes a region-based type and effect system for expressing important patterns of deterministic parallelism in imperative, object-oriented programs. It also describes a language called Deterministic Parallel Java (DPJ) that incorporates these features.

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

This paper introdces a framework to create interleaving free models of isolated hierarchical parallel computations. Since, this framework applies to task parallel programming languages, it can be used to model Habanero Java language. This paper also discusses the complexity of deciding state-reachability for finite-data recursive programs. Using this algorithm, we can say that the computing the state reachability for Habanero Java programs is EXPSAPCE-hard.

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

This paper proposes an assertion framework that can be used to specify pairs of program states that can arise due to non-deterministic thread inter-leavings. Such pairs of program states result in a deterministic output in spite of the different parallel schedules.

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

This paper proposes an algorithm called Determin that can infer likely deterministic specifications for a parallel programs for which the programs behave deterministically.

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

This paper introduces the habanero java language and its features. HJ is a task parallel programming language. It provides a set of parallel programming constructs that can be used as extensions to Java programs to take advantage of the multi-core processors.

[8] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

This paper discusses the various language features and characteristics that are essential for parallel programming. It compares the various parallel programming languages and their features. It then introduces the parallel programming language chapel.

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

This paper introduces X10, a task parallel programming language and its features.

[10] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[11] Jack B Dennis, Guang R Gao, and Vivek Sarkar. Determinacy and repeatability of parallel program schemata. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012*, pages 1–9. IEEE, 2012.

This paper presents precise definitions of two closely related properties of program schemata determinacy and repeatability. These definitions establish relationships between data-flow and task-parallel models.

[12] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

This paper describes a static analysis tool called RacerX. RacerX is used for detecting deadlocks and race conditions in complex multi-threaded systems.

[13] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

This paper introduces a data race detector called "Fasttrack" that is fast and precise. It uses an adaptive representation for the happens-before relation to provide constant-time fast paths for common cases without loss of precision.

[14] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

Message Passing Interface is a standard for writing distributed parallel programs that use message passing as a means for sharing information. It is a practical, portable, efficient, and flexible standard for message passing.

[15] Milos Gligoric, Peter C Mehlitz, and Darko 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.

Creating model checking tools for the new parallel programming languages being developed is very complex and requires great deal of effort. Instead, modifying an existing model checker to suit the needs of the new languages is easier and it helps to use all the advanced features that the existing model checker has for testing the new language. This paper describes the efforts needed to modify an existing model checker JPF to verify programs written in the X10 language.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[17] Shams Imam and Vivek 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*, pages 75–86. ACM, 2014.

This paper discusses the java 8 library implementation details for Habanero Java language.

[18] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jfuzz: A concolic whitebox fuzzer for java. In *NASA Formal Methods*, pages 121–125, 2009.

[19] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun 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.

This paper proposes a static data race detector for concurrent C programs that is fast and precise. For a multi-threaded program with asynchronous calls, it first builds a precise context-sensitive concurrent control flow graph (CCFG) based on a flow and context-sensitive (FSCS) points-to analysis. Using this CCFG, they perform a staged data race detection, that involves identifying the shared variables and lock pointers, computing an initial database of race warnings, and finally, pruning away the spurious warnings using a may-happen-in-parallel (MHP) analysis based on computing lock sets and performing thread order analysis.

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

This paper proposes a concept of Universal causality graphs. UCGs are set of all feasible interleavings that a given correctness property may violate. It gives formal method to find out all the possible interleavings which make the parallel program non-deterministic.

[21] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In *ACM SIGPLAN Notices*, volume 47, pages 285–296. ACM, 2012.

This paper introduces a tool called Redex used for creating mechanized semantic models of programming languages. This tool helps to create executable models of programming languages, state and prove theorems, run randomized testing to detect bugs in the language etc.

[22] Li Lu and Michael L Scott. Toward a formal semantic framework for deterministic parallel programming. In *Distributed Computing*, pages 460–474. Springer, 2011.

This paper discusses a formal semantic framework for deterministic parallel programs.

[23] Robert HB Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

This paper gives formal definitions for different race conditions that can occur in parallel programs.

[24] Adrian Nistor, Darko Marinov, and Josep 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.

This paper proposes a technique called "InstaCheck" for checking determinism in parallel programs. It calculates a 64-bit hash of the output. If two program runs with same input produce different hashes, then insta-check reports that the program is non-deterministic.

[25] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

This paper introduces a tool called Eraser for dynamically detecting data races in parallel programs. Data races cause the parallel programs to behave non-deterministically.

[26] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

[27] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. Automatic verification of determinism for structured parallel programs. In *Static Analysis*, pages 455–471. Springer, 2011.

This paper discusses a technique for verification of determinism in parallel programs. It identifies the fragments of the program that execute in parallel. Then, it checks if these fragments perform independent memory accesses using a sequential analysis.

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

This paper discusses a tool for model checking and symbolic execution of Chapel programs.

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis proposal submitted by

Radha Nakade

This thesis proposal has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Eric Mercer, Chair	Date
Quinn Snell	Date
Parris Egbert	Date
Quinn Snell, Graduate Coordinator	Date