# 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

April 22, 2015

**Abstract**

A computation graph of a task parallel program is a directed acyclic graph that represents the execution of the program. A computation graph can be used to locate data-races in parallel programs. It can also be used to identify meaningful thread schedules in programs with data-races to determine the determinacy of output in such programs. This proposal outlines the implementation details of the computation graph builder and analyzer for task parallel programs using Java Path Finder. It also gives a list of benchmarks that will used to validate the correctness of this approach.

## 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 laptops 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 access patterns are observed. If two or more processes executing in parallel access a common memory location and at least 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-races 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 and reproduce. 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 programs with the help of computation graphs. A computation graph of a program represents the 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 stores the memory locations accessed and updated by each of the operators. It also correctly reflects the control flow structure of the program. These properties are necessary for verification algorithms to correctly identify the concurrency errors in the programs.

## 2 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.

## 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 [16] programs create multiple tasks with each encapsulating some local data. Data is shared between the tasks by sending messages from one task to another. 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 [8] 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 library implementation for this language, known as HJLib [18] 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 by Anderson et al. to use JPF for verifying HJ programs. This runtime provides a lightweight alternative to verifying HJ programs using JPF. JPF [9] is a highly customizable execution environment for verification of Java bytecode programs. It is an explicit-state model checker for Java programs. It collects deep runtime information like coverage metrics. It can be used to detect concurrency errors such as deadlocks, data-races etc. Although, as the program size increases, there is an exponential growth in the size of the state space that needs to be explored by JPF. Hence, JPF is not suitable to verify very-large concurrent systems.

X10 [11] 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 [17] to verify X10 programs and detect concurrency related bugs. Chapel programming language [10] was developed as a part of DARPA's High Productivity Computing Systems (HPCS) program. The Chapel provides higher-level abstractions for parallelism using anonymous threads that are implemented by the 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 [12] 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 [3] . 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 that 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 program on which the error exists. To detect such errors we have to test all the branches of 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.

Task parallel languages achieve parallelism by distributing execution processes across different parallel computing nodes. A formal model is helpful 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 [5]. 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.

**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 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 [13]. 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 [24]. 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 [4]. 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.

Miller and Choi implemented an integrated debugging system for parallel programs in [23]. 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. 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[6]. 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 [7] that can dynamically infer a likely deterministic specification when provided with a set of inputs and schedules. Insta-check [25] 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. The 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 [26] 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 [15]. 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 [14]. 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.The Habanero Java Programming model was built as an extension to the Java-based definition of X10 language. The Habanero Java Library 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. The Habanero Java library creates standard Java class files that can be run on any Java 8 JVM. The Habanero Java library 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 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 = $\langle$V, E$\rangle$ where

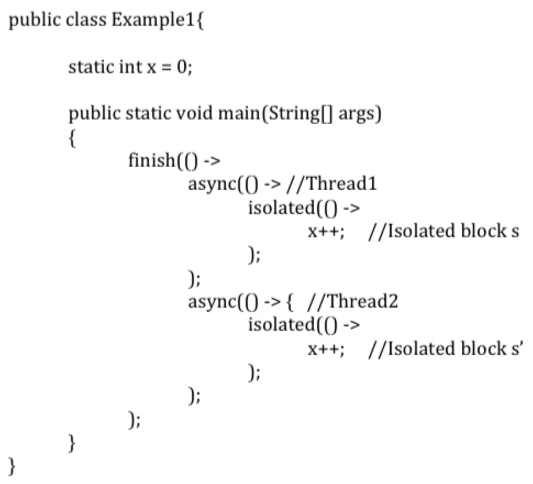
- V is a set of nodes such that each node represents a step consisting of an arbitrary sequential computation and

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

  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 a sample 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. 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 figure 2.



```
public class Example1{

        static int x = 0;

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

Figure 1: Sample HJ Program

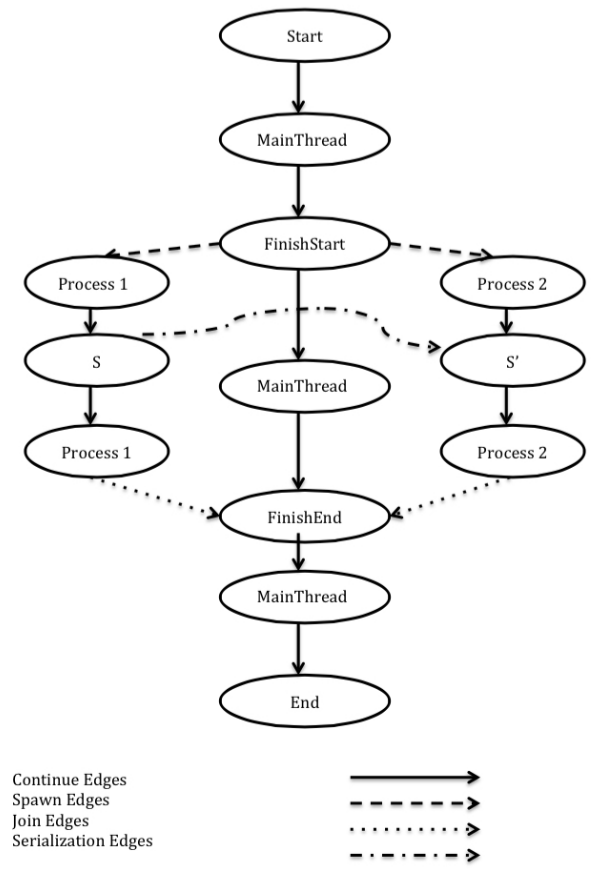


Figure 2: Computation Graph of the Sample program

### 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.

   (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 that execution. The VM listeners in JPF are extended to track the thread creations, executions, joins etc. The core of JPF is a Java Virtual Machine that is used to run the Java bytecode programs. 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 computation graph. The computation graphs are stored in a DAG data structure. We are going to use the 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 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 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 creates thread choice generators to explore different thread schedules that affects the variable that is part of the data race. If all the executions of the program produce the same result, then the program is functionally deterministic.

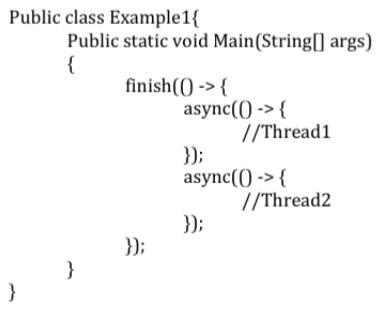
In programs with isolated regions (CG containing serialization edges), JPF creates schedules such that all possible orderings between the interfering isolated nodes is considered. The rest of the process for detecting data-races and checking determinism is carried out on all the obtained computation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

## 5 Validation

In this section we describe the validation process of the claims made in this proposal.

The first claim made in this proposal is that a computation graph is a suitable representation of the execution of task parallel programs. This proposal uses the Habanero Java language to describe the implementation details. Using the semantic model for parallel programming languages described in [5], we can show that Habanero Java is a superset of other task parallel languages. It contains a wide range of parallel programming constructs that can be used to perform all operations that are possible in other task parallel languages such that Cilk, Chapel, OpenMP3.0, X10 etc. Now, in this section, we describe how to create computation graphs for other task parallel languages. The analysis of these computation graphs proceeds in the same way as analysis of computation graphs for Habanero Java programs. The following examples illustrate the computation graphs for programs written in different task parallel languages.

1. **X10**



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

Figure 3: X10 Program

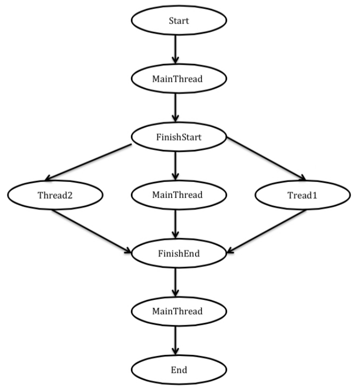
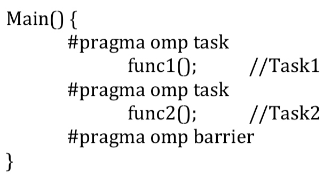


Figure 4: CG of X10 Program

2. **OpenMP3.0**



```
Main() {
        #pragma omp task
                func1();        //Task1
        #pragma omp task
                func2();        //Task2
        #pragma omp barrier
}
```

Figure 5: Openmp Program

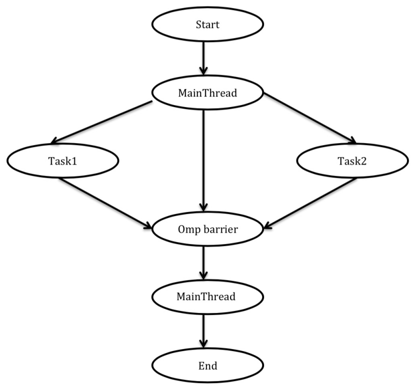
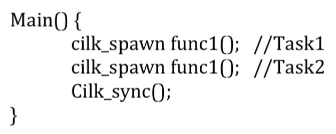


Figure 6: CG of Openmp Program

3. **Cilk**



```
Main() {
        cilk_spawn func1();   //Task1
        cilk_spawn func1();   //Task2
        Cilk_sync();
}
```

Figure 7: Cilk Program

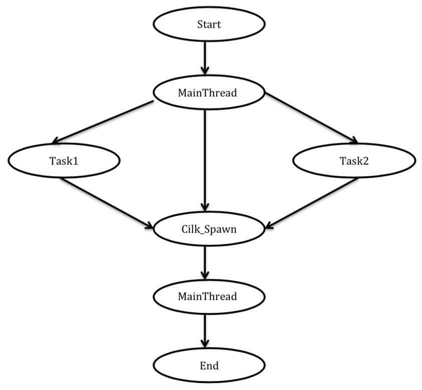
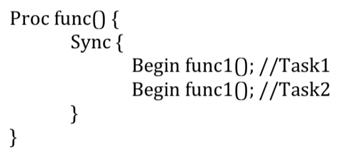


Figure 8: CG of Cilk Program

4. **Chapel**



```
Proc func() {
        Sync {
                Begin func1(); //Task1
                Begin func1(); //Task2
        }
}
```

Figure 9: Chapel Program

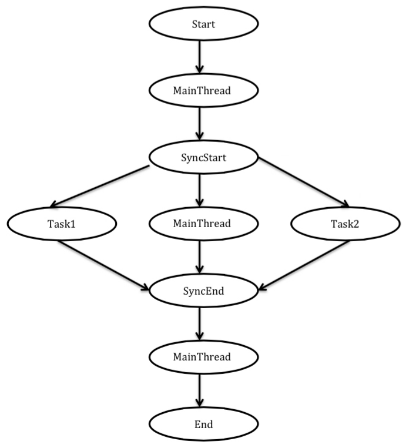


Figure 10: CG of Chapel Program

The second claim is that a computation graph can determine all relevant schedules over tasks that are necessary to be validated to enumerate the possible behaviors of the program. From the implementation details of the computation graph creation, we can see that in the absence of coordination constructs such as 'isolated' that do not impose ordering over the concurrent events or tasks, only one schedule needs to be checked to verify the behavior of the program. In programs involving coordination over tasks, the computation graph marks the events that have to be ordered and inserts serialization edges between them. The analysis of the program takes into account the specific ordering of the events and creates schedules such that each of the event ordering is considered in such programs.

To test the above claim, we have selected benchmarks from the Java Grande Forum benchmarks (JGF) suite, Barcelona OpenMP Task Suites benchmarks (BOTS), Shootout benchmarks suite, and EC2 challenge. These benchmarks include programs that exhibit both kinds of behaviors(single computation graph for programs without co-ordination constructs and multiple computation graphs for programs with co-ordination constructs). The Nqueens and Matrix multiplication benchmarks demonstrate the use of basic parallel programming constructs such as async-finish and loop parallelism constructs such as foreach and forall. The Frannkuch and Mandelbrot benchmarks makes use of the co-ordination construct Isolated in conjunction with the simpler async-finish parallel constructs. The LU-

Fact, SOR and MolDyn benchmarks focus on the co-ordination constructs such as phasers and futures. The BOTS benchmarks from the Barcelona OpenMP Task Suites focus on various parallel constructs of OpenMP. They have been suitably ported to Habanero Java to be verified with this tool. These examples cover all the types of programs that produce different computation graphs. We are going to compute the expected CG structures manually and then compare the output generated by the computation graph builder tool to our expected results. The following table lists out the benchmarks that are going to be used for validation.

| Source | Benchmark | Description |
|---|---|---|
| JGF | Crypt | IDEA encryption |
| | LUFact C | LU Factorization |
| | MolDyn | Molecular Dynamics simulation |
| | MonteCarlo | Monte Carlo simulation |
| | RayTracer | 3D Ray Tracer |
| | Series | Fourier coefficient analysis |
| | SOR | Successive over-relaxation |
| | SparseMatMult | Sparse Matrix multiplication |
| Bots | FFT | Fast Fourier Transformation |
| | Health | Simulates a country health system |
| | Nqueens | N Queens problem |
| | Strassen | Matrix Multiply with Strassens method |
| Shootout | Fannkuch | Indexed-access to tiny integer-sequence |
| | Mandelbrot | Generate Mandelbrot set portable bitmap |
| EC2 | Matmul | Matrix Multiplication |

The third claim made in this proposal is that an exhaustive enumeration of all possible behaviors of a program made with the help of computation graphs is enough for verifying deterministic behavior in task parallel programs. To validate this claim, we use the micro-benchmarks in Habanero Java described in the table below. These micro-benchmarks represent each of the different possible behaviors that can be observed in task parallel programs. A program that is data-race free is always structurally and functionally deterministic. The first example belongs to this category. The second example contains data-races but the program is structurally and functionally deterministic. The third example also has a data-race. The program is structurally deterministic since the program spawns the same number of tasks in every run but functionally non-deterministic because the index of the search term will be different when the searched text has multiple occurrences of the query. The fourth example has data races and is functionally deterministic. But, the program is structurally non-deterministic since number of new spawned tasks depend on the occurrence of the query. The last example has data-races and exhibits both structural and functional non-determinism. This list of micro-benchmarks includes all possible behaviors that a task parallel program can exhibit. The third claim can be validated if the implementation described in this proposal can correctly determine the category for these programs.

| Test Case | Data Race Free | Structurally deterministic | Functionally deterministic |
|---|---|---|---|
| Search Count | Y | Y | Y |
| Existence of an occurrence | N | Y | Y |
| Index of occurrence | N | Y | N |
| Existence of occurrence with no task creation after instance is found | N | N | Y |
| Search Index With No task creation after Instance is Found | N | N | N |

We are going to measure the number of the states explored by this implementation and compare it to the number of states explored by JPF's data-race detector. It is expected that this implementation will explore fewer states to detect data-races. We are also going to compare the runtimes of this method to JPF's data race detector to measure performance improvement.

## 6 Thesis Schedule

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

Submission to Advisor: February 8, 2016
Submission to Committee Members: February 22, 2016
Master's Thesis Defense: April 4, 2016

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

Cilk is a task parallel programming language. It has C-based runtime system. This paper discusses the efficiency of work stealing scheduler of Cilk and various other features of cilk.

[4] 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.

[5] 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 EXPSPACE-hard.

[6] 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.

[7] 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.

[8] 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.

[9] NASA Ames Research Center. *Java Path Finder*, 2005. http:babelfish.arc.nasa.gov/trac/jpf/wiki.

Java PathFinder (JPF) is an explicit-state model checker for Java programs. It is used to verify properties of Java programs. It takes as input a Java program and explores all executions that the program can have.

[10] 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.

[11] 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.

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

OpenMP is a set of compiler directives and callable runtime library routines for Fortran and C to express shared-memory parallelism. OpenMP is designed to be a flexible standard, easily implemented across different platforms.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[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 a 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] Barton P Miller and Jong-Deok Choi. *A mechanism for efficient debugging of parallel programs*, volume 23. ACM, 1988.

This paper describes the use of flowback analysis to provide information on casual relationships between program events by recording only a small amount of trace during program execution. It introduces a mechanism called incremental tracing that makes flowback analysis practical by generating only a small number of traces during execution.

[24] 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.

[25] 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.

[26] 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.

[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 |