Model Checking Task Parallel Programs using Gradual Permission

Eric G Mercer and Peter Anderson
Brigham Young University
Provo, Utah, USA
Email: eric.mercer,anderson.peter@byu.edu

Nick Vrvilo and Vivek Sarkar
Rice University
Houston, TX, USA
Email: nick.vrvilo,vsarkar@rice.edu

***Abstract*—Habanero is a task parallel language that provides correctness guarantees. Even so, programs can be non-deterministic due to deadlock or data-race. This paper presents a verification-specific library for Habanero intended to assist in discovering deadlock and data-race. The library has been extensively tested with Java Pathfinder (JPF) to give confidence in its correctness. This paper further presents an implementation of gradual permission regions in JPF to find deadlock and data-race in Habanero programs. JPF detects sharing, and the user annotates such sharing with permission regions or isolation. Verification only schedules on permission regions or isolation to elicit data-race and deadlock. The approach is sound with respect to proving programs free of such behavior. The results from an empirical study show significant reductions in verification cost, where cost is controlled with the size of the permission regions, at the risk of rejecting programs that are actually free of data-race and deadlock.**

I. Introduction

Despite the explosion in multi-core hardware for general purpose computing, writing programs to take advantage of the available processing power is a task reserved for expert developers. Parallel programming models are nuanced with non-trivial language semantics, and the first programs from the uninitiated have more in common with sequential execution than parallel performance due to excessive synchronization, or worse, those programs are fraught with concurrency errors due to an absence of needed synchronization. Parallel semantics is not the normal mental model for most programmers, and as a result, parallelism is employed little, deployed incorrectly, or exclusively reserved for the expert users which are not found in abundance.

The Habanero extreme scale software research project intends to bring multi-core programming to the masses by providing languages, compilers, run-time systems, and tools to support programmers that are not experts in concurrency. Habanero itself is a task-parallel programming model built around lightweight asynchronous tasks and data transfers. As such, rather than manipulating processes, threads, and synchronization for concurrent execution, the programmer identifies sections of the program that can run concurrently as tasks using simple annotations in the sequential code. An implementation of Habanero would then shoulder the complexity of the parallel execution and absolve the programmer of that responsibility. The programmer now focuses on the high-level task constructs while an implementation worries about how to correctly implement and synchronize those constructs.

Aside from the simplified task-parallel programming model, Habanero gives some limited correctness guarantees. It defines safe subsets of the language that preserve correctness in regards to concurrent interactions. For example, programs that only create tasks and join on their termination are free of deadlock, support serialization (i.e., removing all the annotations yields a sequential program that gives the same computation), and in the absence of data-race (i.e., conflicting concurrent accesses to shared memory), those same programs are deterministic. In a safe subset, a programmer does not need to worry about concurrent interactions between tasks beyond data-race.

Habanero Java (HJ) is the most widely deployed implementation of the Habanero model, and it has been adopted as a pedagogical language for teaching concurrency [**?**]; however, there is a gap between the theory of the language with its safe subsets and the implementation in regards to test and validation. When operating within a safe subset of the language or outside for performance, there is no easy way to determine when and if a program is free of data-race—a necessary condition for determinism. Even debugging computation is non-trivial as the HJ implementation is complex, so a user has no obvious method to track a task, let alone control its execution, using a conventional debugger. As a result, inefficient code inspection, run-time failures, and *printf*-debugging are the primary techniques for test and validation.

This paper presents research to address debugging, test, and validation for task parallel programming models such as Habanero. The first contribution is a new implementation of Habanero for Java in the form of a library (HJ-V). The implementation trades performance for simplicity and correctness. It does this by using Java threads for each task, and using global locks with conditions for features of Habanero that require mutual exclusion and complex synchronization. As such, it is well suited for test and validation since a conventional debugger is able to inspect and control the execution of tasks (e.g., Java threads). Additionally, weighing in with only 32 classes and around 1,300 lines of code, the library is an order of magnitude less complex than even the most simple implementations in the HJ distribution. Careful manual inspection of the code base, which is conveniently small, with extensive testing and verification, reasonably ensure a high degree of confidence in its correctness and supports the claim that HJ-V preserves all behaviors allowed by the Habanero semantics. That said, finding deadlock and data-race in an input program is still a difficult challenge as is enumerating behaviors for test in non-deterministic programs.

The HJ-V library enables model checking for HJ programs. Model checking exhaustively enumerates program behavior, and in the case of task-parallel programming models, it reasons over task schedules to prove the absence of errors. The Java Pathfinder model checker (JPF) is able to directly verify freedom from deadlock and data-race in HJ programs using HJ-V as the Habanero implementation because HJ-V employs a one-to-one mapping between tasks and threads. The verification leverages the native support for threads and locks in JPF to automatically explore all possible ways to schedule concurrent tasks. Such an approach is not possible using the other Habanero implementations in the HJ distribution because JPF does not know where to schedule. The model checking is effective for verifying the HJ-V implementation beyond manual inspection and proving small programs correct, but it does not scale to larger programs.

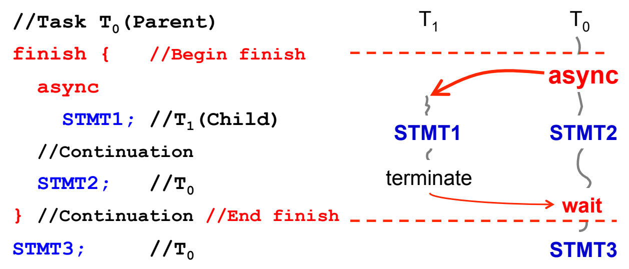
The second contribution to debugging, test, and validation is an implementation in JPF of a sound algorithm for the validation of task-parallel programming models such as Habanero that is able to detect programs that are free of deadlock and data-race (JPF-HJ). It also enumerates all outcomes that arise from non-determinism in sequencing isolated atomic blocks. The algorithm still employs model checking with the HJ-V library as before; however, to scale to larger input programs, the algorithm uses permission regions to annotate atomic blocks of read/write operations on shared memory. These atomic blocks effectively reduce the number of schedules that must be considered to prove a program correct.

Permission regions are program annotations that announce how a task interacts with shared memory (i.e., reading or writing), and over what region of code that interaction takes place [**?**], [**?**]. During execution, auxiliary data structures track access on those memory regions and signal an error on any conflicting access. Permission regions have been shown effective in dynamically detecting data-race at run-time.

JPF-HJ includes an implementation of permission regions and a specialized scheduling algorithm to reduce the number of explored schedules needed to show a program free of deadlock and data-race. The new algorithm only preempts at the entrance to permission regions and isolated atomic blocks to schedule threads. As stated previously, the new algorithm is sound, meaning that it may reject programs that are actually correct if the permission regions are too big, but effective at controlling state explosion. The algorithm does report a witness to any discovered deadlock or data-race violation which can be used to validate the error with the debugger. If the error is a false report due to the size of the regions, then the witness provides insight on how to reduce the size of the permission regions. This new algorithm together with the simplified run-time provide needed support for debug, test, and validation of task parallel programs.

The principle contributions described in this paper are summarized as

- HJ-V: a verification specific implementation of Habanero for Java as a Java library that is extensively tested through model checking with JPF and is amenable to conventional debugging;
- JPF-HJ: an implementation of permission regions in the JPF model checker with a sound algorithm that only schedules on permission regions and isolated atomic blocks to prove a program free of deadlock and data-race; and
- an empirical study showing the impact of permission regions on the complexity of model checking over a set of benchmarks.



```
//Task T0(Parent)
finish {   //Begin finish
  async
    STMT1; //T1(Child)
  //Continuation
  STMT2;    //T0
} //Continuation //End finish
STMT3;      //T0
```

Fig. 1. An example with `async` and `finish`.

The HJ-V library and JPF-HJ specialization are available for download at: http://javapathfinder.org/jpf-hj/.

II. Habanero Programming Model

The Habanero programming model is built around a task-parallel view of concurrency [**?**]. Figure **??** illustrates Habanero in its simplest form [**?**].

The `async`-construct is a mechanism for creating a new asynchronous task: `async` ⟨*stmt*⟩ causes the calling task (i.e., the parent) to create a new child task to execute ⟨*stmt*⟩ (logically) in parallel with the parent task. ⟨*stmt*⟩ can read or write any data in the heap and can read (but not write) any local variable belonging to the parent task's lexical scope. The task created by any `async`-construct is scheduled at the point it is declared in the program.

The `finish`-construct is a generalized join operation for collective synchronization: `finish` ⟨*stmt*⟩ causes the parent task to execute ⟨*stmt*⟩ and then wait until all tasks created within ⟨*stmt*⟩ have completed, including transitively created tasks. Each dynamic instance of a task has a unique *immediately-enclosing-finish* (IEF) during program execution. That IEF is the innermost `finish`-construct containing the task. There is an implicit `finish`-construct surrounding the entry point of the program so the program only terminates after all tasks have completed.

A computation graph illustrating the semantics of the `async` and `finish` constructs is on the right side of Figure **??**. In the graph, task T_0 enters the `finish`-construct, creates task T_1 at the `async`-construct, and then continues on to `STMT2`. After `STMT2`, T_0 waits for T_1 to complete before moving on to `STMT3`. Note that `STMT1` and `STMT2` are not ordered by the semantics and represent parallel execution.

Habanero supports more advanced forms of tasking beyond creation and collective synchronization. The `isolated`-construct, `isolated` ⟨*stmt1*⟩, ensures that ⟨*stmt1*⟩ is evaluated in mutual exclusion with all other `isolated`-constructs. There are two subtle nuances in the Habanero model for the `isolated`-construct:

1) The construct ensures mutual exclusion between `isolated`-constructs and not mutual exclusion on a particular memory location. Mutual exclusion on a particular memory location is implemented by wrapping operations on that memory location in `isolated`-constructs.
2) Any Habanero implementation may relax mutual-exclusion between `isolated`-constructs as long as the constructs do not interfere with one another. Interference in this context means that multiple `isolated`-constructs access a common memory location and at least one of those accesses is a write.

The `future`-construct lets tasks return values to other tasks: **future** f = **async** $\langle expr \rangle$ creates a new child task to evaluate $\langle expr \rangle$. The local variable *f* contains a *future handle* to the newly created task that can be used to obtain the value produced by $\langle expr \rangle$. The blocking operation *f.get()* returns that value when the child task completes.

The most complex construct in the Habanero model is the *phaser* [**?**]. A phaser is a form of a barrier that provides point-to-point fine-grain synchronization between tasks to coordinate their movement through *phases* of computation. Like barriers, phasers order execution of portions of the program into phases and restrict tasks from entering the next phase until the current phase is complete. Unlike barriers though, phasers allow tasks to specify point-to-point relationships on multiple phasers, and tasks can dynamically join or leave the phaser.

Tasks register with an instance of a phaser, and on registration, declare the mode that control how that task synchronizes relative to other tasks registered on the same barrier. Synchronization takes place with the `next`-construct which may block depending on the state of the phaser, and on how the task is registered with the phaser.

- `SIG`: signal registration means all tasks that have designated themselves as signalers must signal the phaser in order for the phase to advance. The `next`-construct for a signal-only task signals the phaser and immediately advances to the next phase. The phaser remembers each phase completed by any task.
- `SIG_WAIT`: *signal-wait* registration means the task signals the phaser and then waits for other tasks to complete the phase. This registration mode functions like a traditional barrier. The `next`-construct for a signal-wait task reports phase completion and then blocks for the other signalers to complete the phase too.
- `WAIT`: *wait* registration means that the task blocks at the `next`-construct until the phase advances.

Phasers may also be bounded to specify slack in the number of phases that may separate waiters and signalers so signalers can work ahead of waiters up to a bound.[1]

Habanero includes several other constructs such as `foreach`-constructs, `forall`-constructs, *data driven futures*, *actors*, etc. most of which are syntactic sugar for the presented constructs.

```
public static void main(final String[] argv) {
  Stack stk = initStack();

  launchHabaneroApp(() -> {
    finish(() -> {

      async(() -> {
        stk.push(5);
      });

      stk.peek();
    });
  });
}
```

Fig. 2. An HJ program snippet using the `async` and `finish` statements and also showing how to start the Habanero environment.

```
public static double
parArraySumFutures(final double[] X) {
  final HjFuture<Double> sum1 = future(() -> {
    // Return sum of lower half of array
    double lowerSum = 0;
    for (int i = 0; i < X.length/2; i++) {
      lowerSum += 1 / X[i];}
    return lowerSum;});

  final HjFuture<Double> sum2 = future(() -> {
    // Return sum of upper half of array
    double upperSum = 0;
    for (int i = X.length/2; i < X.length;
         i++) {
      upperSum += 1 / X[i];}
        return upperSum;});

  // Combine sum1 and sum2
  final double sum = sum1.get() + sum2.get();
  return sum;
}
```

Fig. 3. HJ program snippet using the `future` statement to sum an array in parallel.

III. Habanero Java Implementation

HJ-V is a Java library implementation of the Habanero model designed specifically for test and validation. It consists of roughly 1,300 lines of code in 32 classes. Most of the classes address the programmer interface rather than the library internals. Figure **??** is an example of the interface using Java 8 lambdas. The interface in this implementation is identical to other Java library implementations of the Habanero model [**?**], so this library is interchangeable with those libraries.

The `launchHabaneroApp` call is the entry point into the library. Its parameter is a function that defines the Habanero program to run. The `finish` and `async` calls have their usual meanings, and like the `launchHabaneroApp` call, they take functions as their parameter. In the example in Figure **??**, two

[1]Omitted in this presentation of phasers is the ability of a single task to execute constructs after the end of one phase and before the start of the next phase.

```
public static void main(String[] args) {
  launchHabaneroApp(() -> {
    finish(() -> {
      final HjPhaser ph = newPhaser(SIG_WAIT);
      HjPhaserPair mode(ph.inMode(SIG_WAIT));

      asyncPhased(mode, () -> {
        char[] buffer = bufferTwo;
        while (true) {
          produce(buffer);
          buffer = toggle(buffer);
          next();
        }
      });

      asyncPhased(mode, () -> {
        char[] buffer = bufferOne;
        while (true) {
          consume(buffer);
          buffer = toggle(buffer);
          next();
        }
      });
    });
  });
}
```

Fig. 4. An HJ program snippet using a phaser to synchronize a producer and consumer.

tasks are created: the main task at the launch, and a child task on the `async` call. Both tasks interact with a shared stack `stk`. The finish call does not complete until both tasks have completed.

The implementation of the `async` and `finish` constructs uses Java threads and the ability to join those threads. Flattening the lambda function in the Java 8 interface to an anonymous inner class elucidates the structure of the library. That code for the `async` call in Figure **??** using an anonymous inner-class is shown below:

```
async(new HjRunnable() {
  public void run() {
    X.push(5);
  }
});
```

The parameter for the call is an instance of an `HjRunnable` object, and an `HjRunnable` is an extension to the standard Java thread. The `run` method for the thread is specialized in the anonymous inner-class. A programmer may use either syntax with HJ-V: lambda or anonymous inner-class.

Staying at a high-level view of the implementation, tasks are threads with extra information to implement the Habanero model. To support the `finish`-construct, that thread includes the notion of a *finish-scope*. A finish-scope holds references to any child thread created within a `finish`-construct, and a stack of finish-scopes tracks the nesting of `finish`-constructs within a task. When a task is created, it is added to the current running thread's active finish-scope. In this way, when a parent reaches the end of a `finish`-construct, it is able to join on all threads in the current finish-scope. After joining, the finish-scope is popped from the stack making the next outer finish-scope the active scope.

The program in Figure **??** has a somewhat obvious data-race that is unsafe. Data-race intended by the programmer is made safe, or protected, using the `isolated`-construct. For the referenced example program, the access to the `stk` object by the main task should be wrapped as follows:

```
isolated(() -> {
  X.peek();
});
```

The access in the child task should be wrapped similarly. The two access are now purely sequential being run in mutual exclusion. The `isolated`-construct serializes atomic blocks of code relative to other isolated blocks.

The implementation of the `isolated`-construct is trivial. Every task has access to a `RunTime` object created with the call to `launchHabaneroApp`. That object contains a single lock that is used to force `isolated` constructs to be sequential atomic statements.

Figure **??** is an example of a program using `future`-constructs. The program creates two future tasks to sum the lower and upper halves of an array in parallel. The parent task uses those future tasks, blocking until they complete, to combine the two sums into a final result.

The implementation of the `future`-construct is again accomplished with a thread. The thread is similar to the thread for the `async`-construct only it has the added ability to suspend. Suspension is made possible using a condition on a Java lock. A caller to the `get` method blocks if the function passed into the `future`-construct has not yet completed. Once the function completes, the lock condition is signaled to free any threads blocked in the associated `get` method.

The example in Figure **??** is more complex utilizing a phaser to coordinate a producer task and consumer task. Unlike the other constructs, the `asyncPhased` call takes two arguments: the mode for the single participating phaser (or a list of modes for each of the participating phasers), and a function defining the body of the task. The phaser generates the objects to indicate modes using the `inMode` method. There are 3 tasks in this example. The main task creates and registers itself with the phaser in the `SIG_WAIT` mode. It then creates the producer and consumer tasks. When the main task reaches the end of the `finish` call, it automatically de-registers with the phaser and blocks for the producer and consumer tasks. Each call to `next()` interacts with every phaser passed in on the `asyncPhased` call, so all phasers associated with the task synchronize at the same program location.

As before, the actual task interacting with the phaser is a Java thread. Similar to how `finish`-constructs are implemented, the extra information in the thread includes references to the phasers for the task. The call to `next` iterates two times over the phasers. The first iteration signals as appropriate, and the second iteration waits as appropriate (possibly blocking along the way). The phaser itself uses a Java lock with conditions to track threads that signal, threads that wait, to

```
<T> void acquireR(T xs)
<T> void acquireR(T xs, int idx)
<T> void acquireR(T xs, int start, int end)

<T> void releaseR(T xs)
<T> void releaseR(T xs, int idx)
<T> void releaseR(T xs, int start, int end)
```

Fig. 5. The permission-region annotation interface for read acquisition and release in JPF.

block waiting threads as needed, and to keep track of the actual phase.

Other constructs in the Habanero programming model are implemented similarly. The direct mapping between tasks and threads in the library makes a conventional debugger more feasible for understanding computation as well as for debugging deadlock and data-race. The debugger is able to directly control the scheduling of concurrent threads on a single processor, and because the run-time is relatively small, it is possible to step through the run-time to understand its functionality if needed. As an observation, there is a significant overhead with creating threads for each task, the least of which impacts run-time performance. The intent is to debug with HJ-V and use the performance oriented implementations for deployment.

Aside from conventional debugging, the library enables direct model checking using JPF. JPF is an extensible implementation of a Java virtual machine written in Java [?]. Out of the box JPF is able to model check Java programs for exceptions, assertion violations, deadlock, and data-race. Model checking brute-force explores all possible thread schedules and reports any violating schedule. Although model checking is expensive, it is effective for reasoning over schedules to harden multi-threaded programs. The HJ-V implementation has been extensively model checked by JPF using a set of test input programs. The test input programs explore the different constructs in the Habanero model in an effort to elicit unexpected behavior. The model checking has been extremely helpful in finding bugs, removing deadlock, and eliminating data-race both in the library implementation itself and the input programs.

Model checking does not scale to larger programs, as expected, and JPF is no exception to that rule. A common approach to state explosion though is to reduce the number of schedules that need to be checked in model checking. This research leverages permission regions to define atomic blocks so that JPF does not context switch threads at every access to shared memory. Rather, it is restricted to only schedule at the entry points of permission regions and `isolated`-constructs.

IV. Gradual Permission Regions

Gradual permissions with permission regions is a hybrid static-dynamic approach to detecting data-race in task-parallel programs [?], [?]. The programmer annotates regions of the program text that access shared objects. Those regions are indicated as accessing shared objects in read mode or write mode. When the program runs, a state machine is associated with each shared object to track access permissions on that

```
public static void main(final String[] argv) {
  launchHabaneroApp(() -> {
    Stack stk = initStack();

    finish(() -> {

      async(() -> {
        acquireW(stk);
        stk.push(5);
        releaseW(stk);
      });

      acquireR(stk);
      stk.peek();
      releaseR(stk);
    });
  });
}
```

Fig. 6. The HJ program from Figure ?? with additional permission region annotations.

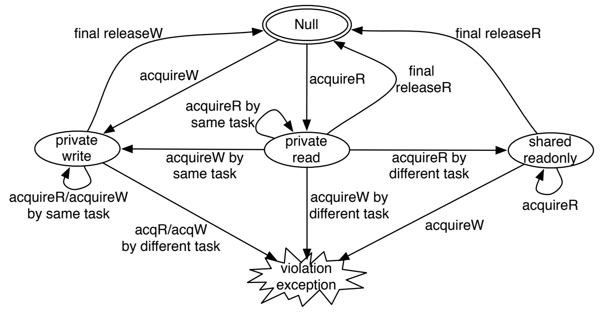


Fig. 7. State machine for permission regions operating on a single object.

object as indicated by the program annotations. If access permissions from distinct tasks on the same object conflict, then a dynamic run-time error is reported.

Permission regions are distinctly different from `isolated`-constructs. Foremost, `isolated`-constructs define atomic regions that run mutually exclusive to other regions in `isolated`-constructs. As such, isolation restricts concurrency by serializing atomic regions. Permission regions do not serialize atomic regions to restrict concurrency. They only check if concurrent accesses to atomic regions are free of data-race. Isolation is always best avoided given its impact on speedup in Amdahl's law.

Permission regions are annotated for JPF using the interface in Figure ??. Although the figure only shows the read interface, the write interface follows the same pattern. The first method on the interface has arguments to manage permissions on a single object. The second and third methods have arguments to manage permissions on arrays. Arrays are somewhat more nuanced. Consider the following code:

```
  f[i] = new C();
  f[i].write(j);
```

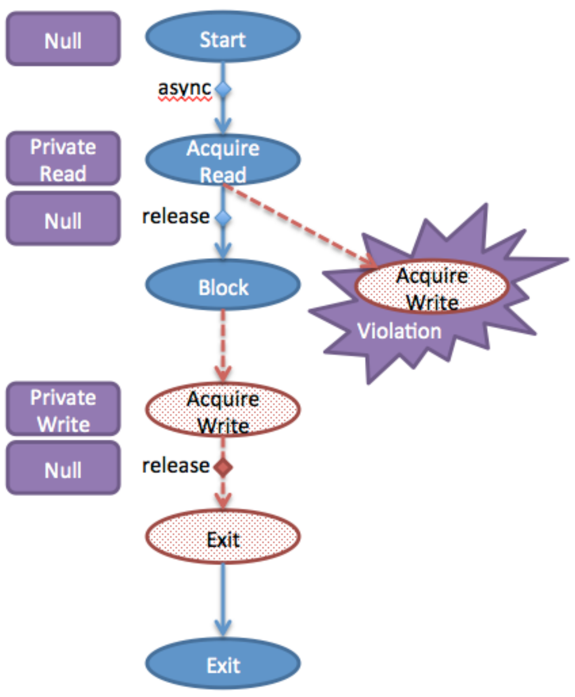


Fig. 8. Different schedules for the program in Figure ?? with the right-most schedule detecting a violation.

The first line writes to the array location i. The second line writes to the object stored in the array location i. These two accesses must be treated separately. As such, the permission regions interface provides methods to manage permissions on a specific index in an array or a range of indexes in the array. Permissions on the actual object in the array use the first method in the interface. The annotated code snippet from above is as follows.

```
acquireW(f,i);
f[i] = new C();
releaseW(f,i);

acquireR(f,i);
acquireW(f[i]);
f[i].write(j);
releaseW(f[i]);
releaseR(f,i);
```

Figure **??** is the permission-region annotated version of the program in Figure **??** from the previous section. The operations on the shared object `stk` are now wrapped in calls to acquire or release permissions on the shared object. Note that the permission regions span all the code in the `stk.push` and `stk.peek` methods, so anytime the object is referenced, it is covered. In this way, permission regions can be as large or as small as desired. If the regions are too large, however, then the approach may report a data-race where no race exists [**?**], [**?**].

Figure **??** is the state-machine to track permissions on shared objects and detect violations. That machine is included here for convenience directly from [**?**]. The machine starts in the double-circled *Null* state. On acquisition or release, the machine updates to the appropriate state based on its current state. The machine signals a violation if it ever detects conflicting accesses by different tasks.

Figure **??** shows two possible schedules for the annotated program in Figure **??**. The solid filled ovals and solid lines represent the main task and the dotted filled ovals and dashed lines represent the task created by the `async`-statement. The squares indicate the current state of the state machine that is tracking accesses to the shared object `stk`.

The left branch of the tree is the schedule where the main task runs until it is blocked by the `finish`-statement where it must join with the tasked created by the `async`-statement. The main task acquires and releases private read privileges on the region before it blocks. After the main task blocks, the newly created task runs, acquiring and releasing private write privileges, and it then exits. If this schedule is followed in the run-time, then no violation is reported even through data-race exists in the program. The approach is run-time dependent and not exhaustive.

The right branch is an alternate schedule that is possible in the program. In this schedule, the newly created task from the `async`-statement runs just after the main task acquires private read privileges on the shared object. When the new task tries to acquire write privileges, the state-machine that manages permissions on the shared object moves into the violation state to report the error.

A. Permission Regions in JPF

The implementation of permission regions in JPF spans 1036 lines of code and covers 11 distinct class objects. It leverages JPF's ability to track thread IDs of all accesses to objects, so it not only reports violations on the permission regions, but also identifies shared accesses that are not annotated by permission regions or covered by `isolated`-constructs. In this way, JPF updates the user when a shared access has been missed in the annotations.

The implementation uses two key features of JPF: byte-code listeners and object attributes. It installs a byte-code listener to watch for instances of the `INVOKE`-code. The actual methods for the permission regions interface in Figure **??** are empty stubs. When the listener sees an instance of the `INVOKE`-code that calls a method on the interface, it gets the method's parameters from the stack and updates the associated state-machines appropriately.

The state machines themselves reside in an attribute of the object. Every object in JPF has an associated attribute that can hold arbitrary information. For example, attributes are used to implement symbolic execution in JPF [**?**]. The important property of attributes is that they follow heap objects through the entirety of state space exploration. The state machines to track permission region accesses are stored in those attributes. For arrays, a separate permissions state-machine is stored for every index. The program annotations acquire and release permissions on individual indexes (or a range of indexes) as mentioned previously.

With or without permission regions, JPF finds the data-race in Figure **??** using its built-in precise-data-race listener with HJ-V. Unfortunately, JPF times-out on larger programs due to state-explosion as shown in the results section. Permission regions are utilized to improve this limitation.

Algorithm 1 Permission Region Informed Search

```
function SEARCH($t$, $h$, $T$)
    loop: ($h$, $T$) := run($t$, $h$, $T$)

    $s$ := status($t$, $T$)
    data-race = false
    if $s$ = PR_ENTRY then
        ($h$, $T$, data-race) := acquire($t$, $h$, $T$)
    else if $s$ = PR_EXIT then
        ($h$, $T$) := release($t$, $h$, $T$)
        goto loop
    end if

    if data-race then
        report data-race and exit
    end if

    $R$ = runnable($T$)
    if $R = \emptyset$ then
        if blocked($T$) $\neq \emptyset$ then
            report deadlock and exit
        else
            report any detected sharing and exit
        end if
    end if

    if $(h,T) \notin S$ then                  ▷ $S$ is a global variable
        $S = S \cup \{(h,T)\}$
        if $s$ = PR_ENTRY $\vee$ $s$ = ISOLATED then
            for all $t_i \in R$ do
                search($t_i$, $h$, $T$)
            end for
        else
            $t_i$ := random($R$)
            search($t_i$, $h$, $T$)
        end if
    end if
end function
```

V. JPF-HJ Search Algorithm

The default model checking algorithm in JPF is too fine grained to scale beyond small programs even in the specialized HJ-V implementation. The cause of the state explosion is JPF's default scheduling algorithm that interleaves the execution of all threads at every thread event and on every access to shared state. For example JPF schedules every thread that can run at every lock acquire, lock release, thread block, thread unblock, field access on shared objects, etc. to exhaustively enumerate the program schedule space. As the HJ-V library relies on locks to synchronize its internal data structures, the fact that tasks are mapped directly to threads, and the fact that often there are several byte-codes that access an object when it is shared, the state explosion is severe.

Permission regions create natural scheduling boundaries for JPF that can be leveraged to mitigate state explosion while preserving the essential behaviors of the program that lead to deadlock or data-race. The intuition is that given a fixed program input, behavior is only affected by interactions between tasks on shared memory. As such, it is only necessary to preempt running threads at the entrance to permission regions and `isolated`-constructs. If a program has deadlock or data-race, such deadlock and data-race exists in one of the schedules that is explored from those preemption points.

Algorithm 2 Procedure to Validate a Program

```
procedure VALIDATE($p$)
    ($h$, $T$) := init($p$)
    $R$ := runnable($T$)
    $t$ := random($R$)
    $S := \emptyset$
    search($t$, $h$, $T$)
    while JPF reports sharing do
        Add permissions regions or isolation for sharing
        ($h$, $T$) = init($p$)
        $S := \emptyset$
        search($t$, $h$, $T$)
    end while
end procedure
```

Algorithm **??** is the pseudo-code for the algorithm to explore all thread schedules created at entry to permission regions and `isolated`-constructs. The state of the Java virtual machine in the pseudo-code is simplified for clarity; it is represented by a heap, h, and a set of threads, T. The lowercase t indicates a thread ID. Line **??** updates the heap and pool of threads by running thread t until it blocks, exits, reaches a permission region boundary (i.e., entry or exit), or reaches an `isolated`-construct.

At the entry point of the permission region, Line **??** updates the state machine for the acquired object in the heap and checks to see if the acquisition signals a data-race. At the exit point of the permission region, Line **??** updates the state machine for the released object in the heap, and the algorithm restarts thread t running anew at Line **??**.

Data-race is reported on Line **??**. Line **??** reports deadlock. A deadlock state is indicated when there are no runnable threads (i.e., $R = \emptyset$) and there exists threads that are blocked. A report for either data-race or deadlock includes a witness trace for validation and debugging. In the absence of deadlock or data-race, and when there are simply no more threads to run, Line **??** terminates the search and reports any detected sharing that was not annotated by a permission region or covered by an `isolated`-construct.

The set S on Line **??** is a global set to track the visited states. Line **??** does the actual scheduling by considering all possible runnable threads, including the currently running thread t, as a next thread to run. Note that in the current state, if the thread t was preempted because it entered a permission region, then that state reflects the acquired permissions on that region. In the case that thread t has been blocked, Line **??** chooses a random runnable thread to schedule next.

Figure **??**, shown previously, is the state space explored by the search algorithm for the simple example Figure **??**. Recall that the example has two tasks that access a shared stack: one reading and the other writing. The ovals in the diagram represent scheduling points, and as before, the blocks represent the state of the machine tracking permissions. As indicated by the pseudo-code, the algorithm only preempts running threads at the entrance to permission regions. In this example, it schedules the child task after the main task acquires read permissions to elicit the violation. By observation, if the permission regions in the annotated program in Figure **??** were replaced with `isolated`-constructs, then the explored state space would no longer include the violation, but it would include another schedule that interleaves the atomic blocks defined by the isolated regions.

Algorithm **??** is a procedural flow describing the process of program validation using the new search in Algorithm **??**. The process leverages JPF's ability to track thread IDs of accesses to heap locations. When the Algorithm **??** finishes, JPF reports any heap locations that have been accessed by more than one distinct thread outside a permission region or an `isolated`-construct with the input program location where that access occurred. Using this information, a user is able to appropriately annotate that program location, and then repeat the search. The process terminates when a deadlock or data-race is discovered, or no more sharing outside of permission regions or `isolated`-constructs exists.

Theorem 1. *The algorithm in Algorithm* **??** *is sound in that it only accepts programs that are deadlock and data-race free for programs that terminate and that have all sharing annotated with permission regions or wrapped in* `isolated`*-constructs.*

Proof: The soundness proof reasons over a slightly modified version of the algorithm that is iterative and takes as an additional input a search tree which is similar to Figure **??** that captures all possible sequences of release and acquire statements explored thus far. The algorithm traverses that input tree and at each leaf node tries to extend that node by one generation if possible. After the traversal, the algorithm returns the new tree. The algorithm is called in an iterative manner until the tree reaches a fix-point (which is guaranteed since the program terminates).

Let $P(n)$ be the statement that this modified search algorithm returns all interesting sequences of acquire and release statements of length n or less for a given input program, where interesting means containing a deadlock or data-race.

Basis Step: the algorithm produces all interesting sequences length $n \leq 1$. This case is trivially established with the initial state of the program that represents a sequence of length $n \leq 1$ and cannot contain a deadlock or data-race since the program has not yet done anything. As such, it includes all interesting sequences.

Inductive Step: assume the modified algorithm has correctly generated a tree representing all interesting sequences of n or less; it is necessary to show that from that tree it is able to generate all interesting sequences of length $n+1$ or less. There are three possible outcomes at any leaf of the input tree:

1) the leaf cannot be extended because it is already an interesting sequence having a data-race or deadlock;
2) the leaf cannot be extended because there are no more runnable threads, in which case it is not interesting; or
3) the leaf is able to be extended with one or more immediate descendants.

The first two cases are directly covered by lines 13 through 24 of the algorithm; there is no way to have any descendants in those situations and the sequences are already classified as interesting or not.

For the third case, first consider line 28 of the algorithm that creates the next generation in the tree for permission regions and isolated blocks. Every runnable thread is scheduled and each of those threads must reach an immediate successor which may be a deadlock or data-race making an interesting sequence, a preemption, a block condition, or exit by the constraint that the input program must terminate. As such, any $n + 1$ length sequence that exists, is generated.

Further, any interesting $n + 1$ sequence is generated as well. To see this outcome, it is important to understand that the order of acquisition relative to read or write does not matter in detecting a violation. The state machine in Figure **??** is not affected by acquisition order; it is only dependent on what read or write permissions are held at the time of acquisition. As the algorithm always first acquires a permission and then schedules other threads, it generates all the interesting $n + 1$ sequences if any exist that are interesting due to data-race. If data-race does not exist, then deadlock is detected as usual.

To complete the inductive step, line 32 must be considered, which covers a blocked or exited thread. The input program has all sharing annotated or isolated by constraint, meaning that any non-determinism due to scheduling is covered already by line 28 so all reachable program paths are considered. If an interesting sequence exists because of deadlock, then it is either found in the $n + 1$ step, by having selected the correct thread, or in a later step when the correct thread is chosen. If the deadlock depends on a particular sequence of thread executions, then those sequences are enumerated by line 28. As such, the deadlock is either deterministic (i.e., independent of the schedule) or non-deterministic (i.e., a product of data-race on some shared object). In the former, the choice of thread does not matter, and in the latter, line 28 enumerates all possible orders over isolated blocks and permission region blocks that do not data-race. ■

A. JPF Implementation

The JPF implementation of Algorithm **??** exploits another aspect of the extensible nature of the tool by providing a new *scheduling-factory*. A scheduling-factory is activated on preemption, when a thread is no longer able to run, or if there is input non-determinism. JPF uses scheduling-factories to decide what threads are scheduled by having the scheduling factory insert *choice-generators* into the state search. The choice-generator enumerates the available choices, and the search iterates over those choices starting a new search for each choice.

The default scheduling-factory of JPF is replaced with a new factory that does not insert any choices on thread actions, locks, synchronization, or shared access to objects. Anything related to concurrency is turned off by the new scheduling-factory except for forced context switches such as a thread exiting or a thread blocking. In those cases, the new scheduling-factory inserts a choice generator with a single choice that represents a random thread that is runnable.

To insert the preemption points for permission regions and `isolated`-constructs, the byte-code listener from the implementation of permission regions is extended to also listen for the `INVOKE`-bytecode calls to `isolated`. At the entrance to permission regions, the permission regions' state-machine for the object is updated as before, but after the update, a choice-generator is inserted into the search that includes choices for all runnable threads. Similarly, a choice-generator is inserted at the `isolated` call. The entire implementation is only a few hundred lines of code but has a significantly reduces the verification cost.

VI. Results

To verify the correctness of HJ-V, test cases were created that utilize specific features of the runtime. Each of these test cases are run within JPF with full scheduling enabled. Thus, for each case, JPF is used to determine that HJ-Vis free of deadlock/data-race. In total 22 test cases were created consisting of approximately 1000 lines of source code.

For comparison between JPF's PreciseRaceDetector and permission regions a series of measurements is collected that perform work in parallel. These benchmarks contain a wide variety of HJ features, including: async, isolated, finish, futures, and phasers. Many of these benchmarks also include the use of shared arrays.

The Error Note column describes the result of the verification attempt. A note accompanied by an * signifies that the operation was incorrect (i.e. race is reported for a program that is free of race).

Each benchmark was run for up to 30 minutes. After 30 minutes the program was terminated and the result is reported as N/A. As table **??** shows, JPF was unable to complete in time for many of the benchmarks. This highlights the challenge of trying to model check all but the most basic of programs. JPF-HJ performs exceptionally well on the following classes of programs: programs without shared state and programs that use arrays to store shared state, but generally access disjoint portions of the array. For the latter case JPF-HJ performs a scheduling optimization. If it can be easily determined by the user that the access to the array will be performed on disjoint portions than a modified form of the permission regions annotation may be used to signal to the runtime to insert just a single scheduling point instead of a scheduling point for each array access. If the programmer is incorrect is their assessment then JPF-HJ will claim the program contains a race when in reality it may not.

It is challenging to predict how the use of specific concurrency primitives will affect the size of the state space. Certainly, shared state produces a larger state space, because of the need to schedule on permission region boundaries. PrimeNumCounter and its variants highlight the effect of shared state on the verification problem as seen in listing **??**. Although this program only has a single piece of shared state it is shared between all 15 tasks.

JPF-HJ also shines when many accesses are performed on a single piece of shared state in a row as seen in listing **??**. This program is simple case for JPF-HJ to verify because only a single scheduling point is needed for every access on shared state. This is in contrast to vanilla JPF which would insert a scheduling point at each bytecode access.

```
public class PrimeNumCounter {
  private final static int COUNT = 17;
  private static int[] primes = {0};
  public static boolean isPrime(int num) {
    //Determine if number is prime
  }
  public static void main(String[] args) {
    launchHabaneroApp(() -> {
      finish(() -> {
        for (int i = 2; i < COUNT; i++) {
          final int j = i;
          async(() -> {
            if (isPrime(j)) {
              isolated(() -> {
                acquireW(primes, 0);
                primes[0]++;
                releaseW(primes, 0);
              });
            }
          });
        }
      });
    });
  }
}
```

Fig. 9. A Naive Method to Count Prime Numbers in Parallel

In summary, JPF-HJ shows a great deal of progress over JPF for a specific class of programs. For the programs for which a side-by-side comparison can be made, Table **??** shows that JPF-HJ 's optimizations reduce the state space by at least two orders of magnitude (PrimitiveArrayRace, PrimitiveArrayNoRace, VectorAdd, etc).

VII. Related Work

There is an existing extension for JPF for the X10 Language [**?**], [**?**]. Habanero is closely related to X10 in many of its constructs. In the extension, JPF operates directly on the actual X10 runtime system. To accomplish the integration, JPF is modified, the X10 runtime is modified, the X10 compiler is extended, and a new static analysis is presented to help control state explosion. The extension represents a significant effort that affects all aspects of the X10 framework to enable JPF verification.

There is a formal model for the Chapel language with an accompanying model checker that employs symbolic execution [**?**]. The formal model is an intermediate representation (IR) suitable for concurrent constructs. The approach compiles Chapel programs into the IR and the model checker then verifies the IR for deadlock and data-race freedom. Creating a compiler and model checker is a significant undertaking beyond the approach in this paper. More critically, the verification tool models the runtime including the number of available worker threads to service tasks; thus, the verification results are dependent on the number of worker threads in the configuration rather than the semantics of the Chapel language.

```
public class ClumpedAcess {
 private static final int[] shared-state =
     {0};
 public static void main(String[] args) {
   launchHabaneroApp(() -> {
     async(() -> {
       isolated(() -> {
         acquireW(shared-state, 0);
         for (int i = 0; i < 1000; i++) {
           shared-state[0]++;
         }
         releaseW(shared-state, 0);
       });
     });
     async(() -> {
       isolated(() -> {
         acquireR(shared-state, 0);
         for (int i = 0; i < 1000; i++) {
           System.out.println(shared-state[0]);
         }
         releaseR(shared-state, 0);
       });
     });
   });
 }
}
```

Fig. 10. Block Access vs. Bytecode Access

Another approach to verifying concurrent languages is to leverage the production level language runtime system itself [?], [?], [?], [?]. These approaches typically require instrumentation of the source program, wrappers to intercept calls into the runtime, and a way to control runtime behavior. Although they are typically able to generate states faster than JPF, verification results are dependent on the employed runtime correctly implementing the language semantics.

Many tools for dynamic race detection have been developed [?], [?], [?]. These tools track the set of locks held by each task during execution and use these sets to determine if a shared resource is insufficiently protected. These tools produce results that are independent of thread interleavings. This is an improvement as compared to previous tools that are dependent upon the thread interleavings of the current execution [?], [?], [?]. Race detection algorithms have also been developed for task-parallel languages [?], [?]. These approaches utilize the structured parallelism of the language to quickly detect races. However, the results of this approach are also limited to a single execution.

Many different approaches have been developed to statically detect race conditions in programs [?], [?], [?], [?]. Each of these techniques require varying levels of instrumentation by the user. RacerX infers the resources each lock protects, code contexts which are multithreaded, and race conditions that have a ”dangerous” effect upon the running program. RacerX relies upon the user to annotate the location of the method for performing the lock/unlock operations. Any other annotations by the user are acceptable, but not required.

Relay performs a static lockset analysis. The Relay algorithm computes relative locksets that belong to each function in the program. This bottom-up approach scales very nicely, however, this approach remains unsound.

General permission regions (GPR) is another static analysis strategy that infers the locations in which to place program annotations [?]. Once the annotations have been statically injected, a dynamic analysis is run to detect the presence of race conditions. Unlike, RacerX, GPR doesn't require any user annotations, although it will honor any annotations introduced by the user. GPR correctly detects race conditions for most common parallel programming paradigms.

Recent work proves the problem of state-reachability to be decidable and EXPSPACE hard for finite-valued programs in languages such as X10/Habanero [?]. The result is limited to a subset of the powerful task constructs in such languages and justifies a model checking effort. The computability and complexity of the more advanced constructs such as phasers is yet to be determined.

VIII. Conclusions & Future Work

This paper presents an approach for the test and validation of task-parallel languages using the Habanero programming model as an example. The approach creates a specialized implementation of Habanero that is purposed for validation. The implementation is not only much simpler than performance oriented implementations, it facilitates a conventional debugger in controlling the scheduling order of concurrent tasks. More importantly, the implementation lends itself to model checking to prove that an input program is free of deadlock and data-race. As model checking does not scale to larger programs, this paper presents a sound algorithm for proving a program free of deadlock and data-race that uses permission regions to mitigate state explosion. The algorithm reduces the number of schedules it must consider by only preempting at the entrance to permission regions. If the regions are too large though, the algorithm may reject a program that is actually free of deadlock and data-race. Since the model checker provides a witness to any detected violation, it is possible to manually validate the witness to refine the permission regions as needed. The approach is illustrated with a full implementation in the JPF model checker and results on several input programs.

Future work includes automating the annotation of permission regions based on the sharing detection in JPF; automating the validation of the counter-example to see if it is real or an artifact of the permission regions being too big; developing techniques that use the counter-example to automatically refine permission regions; and incorporating into the verification process static-analysis to prevent scheduling on regions that statically cannot race.

TABLE I. BENCHMARKS OF HJ PROGRAMS: JPF-HJ VS. PRECISERACEDETECTOR

			Permission Regions				PreciseRaceDetector		
Test ID	SLOC	Tasks	States	Time	Regions	Error Note	States	Time	Error Note
PrimitiveArrayNoRace	29	3	5	0:00:00	0	No Race	11,852	0:00:00	No Race
PrimitiveArrayRace	39	3	5	0:00:00	2	No Race	220	0:00:00	Detected Race
TwoDimArrays	30	11	15	0:00:00	0	No Race	597	0:00:00	DetectedRace*
ForAllWithIterable	38	2	9	0:00:00	0	No Race	N/A	N/A	N/A
IntegerCounterIsolated	54	10	1,013,102	0:05:53	3	No Race	N/A	N/A	N/A
PipelineWithFutures	69	5	34	0:00:00	1	No Race	N/A	N/A	N/A
SubstringSearch	83	59	8	0:00:00	2	Detected Race	N/A	N/A	N/A
BinaryTrees	80	525	632	0:00:03	0	No Race	N/A	N/A	N/A
PrimeNumCounter	51	25	231,136	0:01:08	2	No Race	N/A	N/A	N/A
PrimeNumCounterForAll	52	25	6	0:00:00	2	Detected Race*	N/A	N/A	N/A
PrimeNumCounterForAsync	44	11	449,511	0:02:51	2	No Race	N/A	N/A	N/A
ReciprocalArraySum	58	2	32	0:00:06	2	No Race	N/A	N/A	N/A
Add	67	3	62,374	0:00:33	6	No Race	4,930	0:00:03	Detected Race*
ScalarMultiply	55	3	55,712	0:00:30	2	No Race	826	0:00:01	Detected Race*
VectorAdd	50	3	17	0:00:00	4	No Race	46,394	0:00:19	No Race
AsyncTest1	23	51	54	0:00:00	0	No Race	N/A	N/A	N/A
AsyncTest2	32	3	4	0:00:00	2	Detected Race	11,534	0:00:04	Detected Race
FinishTest1	32	3	6	0:00:00	0	No Race	2,354	0:00:02	No Race
FinishTest2	33	3	5	0:00:00	0	No Race	25,243	0:00:09	No Race
FinishTest3	44	4	7	0:00:00	0	No Race	34,459	0:00:12	No Race
ClumpedAccess	30	3	15	0:00:00	2	No Race	N/A	N/A	N/A